VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

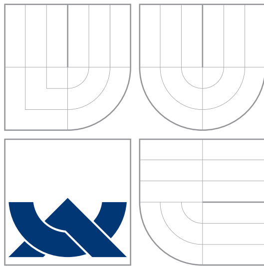
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ V JAZYCE PYTHON

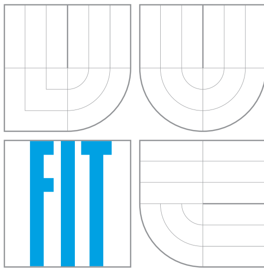
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE PETR DVOŘÁČEK
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KARTÉZSKÉ GENETICKÉ PROGRAMOVÁNÍ V JAZYCE PYTHON

CARTESIAN GENETIC PROGRAMMING IN PYTHON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETR DVOŘÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. ZDENĚK VAŠÍČEK, Ph.D.
SUPERVISOR

BRNO 2013

Abstrakt
Kartézske genetické programování (CGP) patří mezi evoluční algoritmy. Byl primárně vytvořen pro návrhu kombinačních obvodů. Dále může být použit k optimalizaci funkcí, v klasifikaci, evolučním umění atd. Tato práce se zabývá akceleračními technikami urychlující výpočet kandidátního řešení CGP v jazyce Python.

Abstract
Cartesian genetic programming (CGP) is one of the evolutionary methods. It was created for electronic circuit design. It can be used also in optimization of functions, classification, evolutionary art etc. This paper describes acceleration techniques to speed up the evaluation of candidate solution in CGP in Python.

Klíčová slova
Kartézské genetické programování, genetické programování, Python, akcelerace, kompilace za běhu programu, výpočet fitness funkce, symbolická regrese

Keywords
Cartesian genetic programming, genetic programming, Python, acceleration, just-in-time compilation, fitness evaluation, symbolic regression

Citace
Petr Dvořáček: Kartézské genetické programování v jazyce Python, bakalářská práce, Brno, FIT VUT v Brně, 2013

Kartézské genetické programování v jazyce Python

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Zdeňka Vašíčka.

....
Petr Dvořáček
14. května 2013

Poděkování

Chtěl bych poděkovat Zdeňku Vašíčkovi za jeho konzultace a rady spojené s tvorbou této práce.

© Petr Dvořáček, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Kartézské genetické programování **4**
2.1 Evoluční algoritmus . . . 4
2.1.1 Genetické programování . . . 5
2.2 Princip kartézského genetického programování . . . 5
2.2.1 Prohledávací algoritmus . . . 5
2.2.2 Implementace operátoru mutace . . . 6
2.2.3 Fitness funkce . . . 7
2.2.4 Optimalizace . . . 8

3 Python a možnosti akcelerace výpočtu **9**
3.1 Tvorba programu v Pythonu . . . 9
3.2 Běžné metody akcelerace výpočtu fitness funkce . . . 10
3.2.1 Paralelní simulace . . . 10
3.2.2 Akcelerace využívající znalosti fenotypu . . . 12
3.2.3 Kompilace za běhu programu . . . 12
3.3 Využití PyPy pro akceleraci programu v Pythonu . . . 15
3.4 Cython . . . 15
3.4.1 Syntaxe Cythonu . . . 16
3.4.2 Just-in-time kompilace . . . 16

4 Implementace navrženého řešení **17**
4.1 CGP v Pythonu . . . 17
4.1.1 Naivní řešení evolučního návrhu kombinačních obvodů . . . 18
4.1.2 Převod dat do trénovacích vektorů . . . 19
4.1.3 Just-in-time kompilace pomocí funkce `compile()` . . . 19
4.1.4 Just-in-time kompilace do bytecode . . . 19
4.1.5 Just-in-time kompilace do bytecode v postfixové notaci . . . 20
4.2 Cython . . . 20
4.2.1 Tvorba modulu . . . 20
4.2.2 Akcelerace modulu . . . 21

5 Experimentální vyhodnocení **22**
5.1 Verifikace korektnosti fitness funkce . . . 22
5.2 Vliv paralelní simulace na výkonnost . . . 23
5.3 Vyhodnocení rychlosti implementace využívající kompilaci chromozomu . . . 24
5.4 Vyhodnocení rychlosti implementace využívající Cython . . . 28

5.5 Evoluční návrh klasifikátoru . . . 29

6 Závěr **30**

A Obsah CD **32**

B Bytecode **33**

C Tabulky výsledků **35**

D Použití výsledného modulu **38**
D.1 Konstruktor . . . 38
D.2 Změna grafu . . . 38
D.3 Čtení dat ze souboru . . . 39
D.4 Změna dat . . . 39
D.5 Spuštění algoritmu CGP pro symblockou regresi či klasifikaci . . . 39
D.6 Spuštění algoritmu CGP pro návrh kombinačních obvodů . . . 39
D.7 Změna funkční množiny . . . 39
D.8 Výsledky algoritmu CGP . . . 40

Kapitola 1

Úvod

Genetické programování umožňuje generovat celé programy. John Koza jej navrhl v rámci symbolické regrese. Tyto programy byly reprezentovány syntaktickými stromy.

Kartézské genetické programování (CGP z angl. Cartesian Genetic Programming) patří mezi evoluční algoritmy. Tento algoritmus byl vynalezen Julian Millerem a Peter Thomsonem v roce 1999 pro tvorbu kombinačních obvodů, pro které původní genetické programování není tak efektivní. Kandidátní řešení programů v CGP jsou reprezentovány acyklickým orientovaným grafem. Evaluace kandidátního řešení (programu) je však výpočetně náročný proces a je vhodné jej akcelerovat.

Význačnou vlastností jazyka Python je produktivnost z hlediska rychlosti psaní programů. Bohužel interpretování Pythonu evaluaci kandidátního řešení rovněž prodlužuje.

Uplatnění CGP můžeme nalézt také v návrhu čislicových filtrů, obrazových operátorů, kontrolérů pro roboty, při řešení úlohy klasifikace, optimalizaci rovnic, výčet je nekonečný [7]. V této práci se především zaměřuji na aplikaci kartézského genetického programování v oblasti návrhu číslicových kombinačních obvodů a řešením problému symbolické regrese. Algoritmem CGP se zabývám v kapitole 2. Jako akcelerační metoda byla zvolena kompilace heuristické funkce ohodnocující dané kandidátní řešení. Tato funkce se v případě CGP volá nejčastěji. Jedná se o takzvanou Just-in-time (JIT) kompilaci do bytecode (popřípadě do strojového kódu). O této problematice se věnuji více v kapitole 3.2.

V jazyce Python můžeme snadně kompilovat kód za běhu programu. Jistou nevýhodou však je, že se jedná jazyk s dynamickým typováním. Tato skutečnost má za následek, že výsledný algoritmus je obvykle pomalejší než v kompilovaném jazyce. Jazyk Cython rozšiřuje jazyk Python o datové typy a navíc Cython podporuje volání funkcí jazyka C. Pomocí něj můžeme vytvářet moduly a optimalizovat kód Pythonu. Více o jeho použití se můžete dočíst v kapitole 3.4.

V kapitole 4 se věnuji implementaci jednotlivých akceleračních metod. Jejich výsledky naleznete v kapitole 5.

Kapitola 2

Kartézské genetické programování

Kartézské genetické programování jakožto varianta genetického programovaní patří do třídy takzvaných evolučních algoritmů. Tyto algoritmy vychází z Darwinovy teorie evoluce a modernějších evolučních teorií. Podle Ch. R. Darwina je hlavní hybnou silou evoluce *přírodní výběr* (jinými slovy *selekce*) jedinců. Jedinci se od sebe liší svými znaky a přírodní výběr zvýhodňuje nositele těch znaků, které zvyšují pravděpodobnost jeho přežití. Nevýhodné znaky tuto pravděpodobnost snižují. Potomky plodí především nejlépe přizpůsobení jedinci a tím přenášejí své genetické vlastnosti ve zvýšené míře. Přírodní výběr tedy způsobuje to, že přežívají pouze jedinci, kteří jsou nejlépe přizpůsobeni danému prostředí a jsou nejúspěšnější v reprodukci [5].

2.1 Evoluční algoritmus

Evoluční algoritmus pracuje obvykle následovně. Na počátku evolučního algoritmu je vytvořena počáteční populace (množina), která obsahuje zvolený počet kandidátních řešení. Tato populace může být zvolená náhodně, nebo pomocí heuristiky využívající již známá řešení problému. Aplikací evolučních operátorů inspirovaných křížením a mutací na vybrané jedince populace, vznikne množina potomků. Každá nová populace vznikne výběrem kandidátů z předchozí populace a jejich potomků. Tento výběr kandidátů probíhá tak, že se nejdříve ohodnotí všechna řešení heuristickou funkcí zvanou *fitness*. Hodnota fitness funkce udává kvalitu řešení. Při výběru nové populace se vybírají nejen optimální řešení, ale i ty nekvalitní, aby populace nezdegenerovala – neuvázla v lokálním extrému. Algoritmus je ukončen po nalezení dostatečně kvalitního (optimálního) jedince [10]. Samotný evoluční algoritmus pak vypadá následovně:

Algoritmus 1: *Evoluční algoritmus*

```
t = 0 {generace}
P(t) = vytvoření_počáteční_populace
ohodnocení P(t)
while ukončovací_podmínka == FALSE do
   Q(t) = výběr_rodiče(P(t))
   Q'(t) = vytvoření_nových_jedinců(Q(t))
   ohodnocení Q'(t)
   P(t + 1) = výběr_jedinců_do_nové_populace(P(t), Q'(t))
   t = t + 1
end while
```

2.1.1 Genetické programování

Genetické programování (GP) je jedna z variant evolučních algoritmů, při níž můžeme automaticky generovat celé programy. Hlavní podstatou genetického programování je evoluce výpočetních struktur (např. matematické rovnice, elektronické obvody nebo počítačové programy), které jsou reprezentovány stromy. Tyto výpočetní struktury se nazývají chromozomy[1]. Tyto struktury jsou proměnné délky, tudíž může dojít k jejich nadměrnému nárůstu. Tento jev se nazývá *bloat* [7].

2.2 Princip kartézského genetického programování

Nevýhody genetického programování se snaží potlačit tzv. kartézské genetické programování, jehož výpočetní struktury (chromozomy) jsou modelovány acyklickým orientovaným grafem. Každý uzel grafu reprezentuje určitou funkci. Na počátku evoluce se určí parametry tohoto grafu a jeho uzlů: počet primárních vstupů grafu n_i, počet primárních výstupů grafu n_o, počet sloupců grafu n_c, počet řádků grafu n_r. K tomu je zapotřebí určit parametr L (L-back, level back) definující míru propojitelnosti uzlů mezi jednotlivými sloupci grafu. Pro $L = 1$ je propojitelnost minimální, neboť se propojují pouze uzly ze sousedních sloupců. Pro $L = n_c$ je pak propojitelnost maximální – můžeme propojit uzly libovolně. Zapojení primárních vstupů n_i a primárních výstupů n_o může býti libovolné. Dále je nezbytná množina funkcí Γ z níž je přidělena operace uzlu. Zapotřebí je mimo jiné určit maximální aritu operací n_a (maximální počet vstupů do uzlu).

Zapojení tohoto grafu je zakódováno chromozomem **konstantní délky**. Toto je hlavní důvod, proč v CGP nedochází k jevu *bloat*, jehož význam je uveden podkapitole 2.1.1. Jeho délka je dána rovnicí:

$$G = n_c * n_r * (n_a + 1) + n_o$$

Způsob jeho kódování je následující. Každému primárnímu vstupu obvodu se přidělí index z intervalu $0, \ldots, i-1$. K jednotlivým sloupcům se pak přiřadí indexy, a to po sloupcích, s počáteční hodnotou i pro nejlevější horní uzel. Každý uzel obvodu je pak zakódován pomocí $n + 1$ celočíselných hodnot. Prvních n hodnot určuje indexy uzlů, na které jsou pak napojeny vstupy daného uzlu. Poslední hodnota značí funkci uzlu. Na konec chromozomu se přidá posledních n_o hodnot definujících indexy uzlů, na které jsou připojeny primární výstupy.

Z důvodu konstantní délky chromozomu vzniká redundance (jinými slovy nadbytek) uzlů. To znamená, že některé uzly ve finálním obvodu nemusí být vůbec použity, jiné mohou být použity vícekrát. Díky kódování uzly nemusí využívat všechny vstupy. Například při $n_a \geq 2$ hradlo NOT využívá pouze jeden vstup. Dále může nastat situace, kdy skupina uzlů může být nahrazena menším počtem uzlů. Například dva uzly `NOT(AND(A, B))` lze nahradit jedním: `NAND(A, B)`. Tento problém však řeší následná optimalizace.

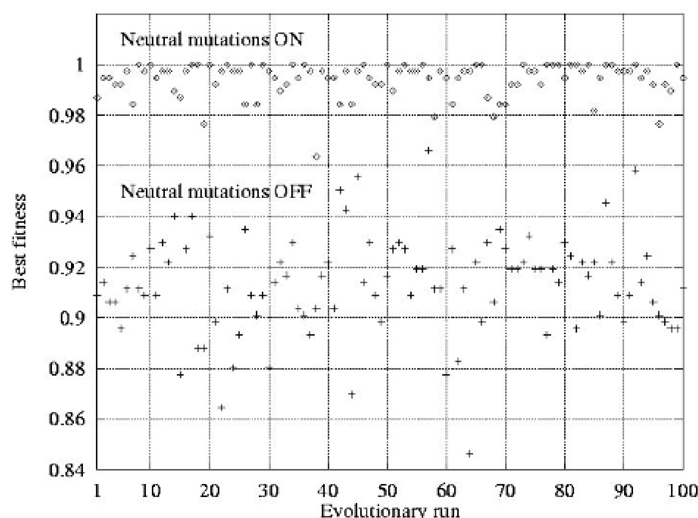
2.2.1 Prohledávací algoritmus

Cílem evolučního návrhu je nalézt takový chromozom, který splňuje specifikaci zadanou uživatelem. V případě evolučního návrhu obvodů se typicky jedná o pravdivostní tabulku a cílem je nalézt takový fenotyp, který na všechny vstupy reaguje přesně podle specifikace. Toho lze dosáhnout jistou variací evolučního algoritmu, který je uveden v podkapitole 2.1.

[1] Chromozom v genetice znamená něco jiného. Z pohledu genetika je to buněčná struktura v jádře, která obsahuje DNA a proteiny. Umožňuje rovnoměrné rozdělení genetické informace při replikaci buňky [5].

Populace se stává z $(1+\lambda)$ jedinců, kde 1 značí právě jednoho rodiče a λ značí jeho potomky. Tato populace se potom ohodnotí. Výběr rodičů je odlišný než u klasických evolučních algoritmů. Vybírá se nejlepší jedinec. Pokud ovšem existuje ve skupině jedinec se stejnou hodnotou fitness, jakou má rodič, vybere se právě ten, který nebyl v předchozí generaci rodičem. To znamená, že se vybírají neutrálně zmutovaní jedinci. Tomu se děje z důvodu genetické diverzity (různosti) jedinců. Tudíž je větší pravděpodobnost, že získáme správné řešení, jak je uvedeno na obrázku 2.1. Evoluce končí nalezením dostatečně kvalitního jedince nebo vyčerpáním počtu generací. Samotný evoluční algoritmus kartézského genetického programování tedy tvoří tyto kroky:

1. Inicializace populace – vytvoření $1 + \lambda$ jedinců.
2. Ohodnocení všech jedinců populace pomocí fitness funkce.
3. Nalezení nejlépe ohodnoceného jedince. (kontrola jeho duplicity na rodiče)
4. Naklonování λ potomků a jejich následná mutace.
5. Nejlepší jedinec s jeho λ potomky tvoří novou populaci.
6. Pokračuje se bodem 2, pokud nebyla splněna počáteční podmínka.



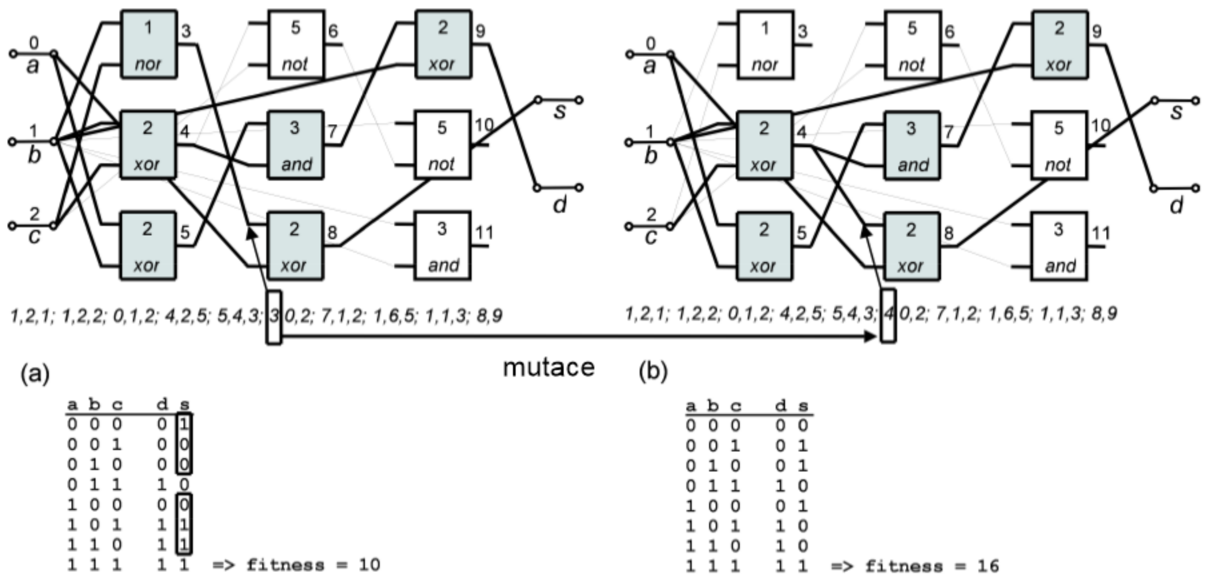
Obrázek 2.1: Neutrální mutace [7].
ON – algoritmus CGP. OFF – vybírají se pouze lepší jedinci.

2.2.2 Implementace operátoru mutace

Mutací během evoluce měníme propojení jednotlivých uzlů, či jejich funkce nebo připojení primárních výstupu obvodu. Aplikací operátoru mutace se pak mohou změnit aktivní uzly na neaktivní a naopak. Tuto situaci ilustruje obrázek 2.2. Obrázek 2.2a ukazuje fenotyp a genotyp před mutací. Obrázek 2.2b ukazuje fenotyp a genotyp po aplikaci operátoru mutace. Rovněž na tomto obrázku můžete vidět, že mutace sama o sobě má velký vliv na výsledek.

Stačí pouze jedna malá změna v chromozomu a můžeme dostat buď velmi kvalitní, nebo velmi nekvalitní řešení. Mutace se nazývá neutrální, pokud nemá vliv na fitness hodnotu. To se může stát v případě, že byla aplikována na neaktivní uzly, anebo že nalezneme řešení s jiným fenotypem, ale se stejnou fitness hodnotou.

Klasické genetické programování využívá oba genetické operátory (křížení, mutace). Operátor **křížení** se v případě CGP **nepoužívá**! Neboť samotnou evoluci zpomaluje a hlavně vede k horším výsledkům.



Obrázek 2.2: Aplikace mutace v CGP

2.2.3 Fitness funkce

Fitness funkce slouží k řízení evolučního algoritmu napříč prohledávacím prostorem. K tomu je nutné každému kandidátnímu řešení (chromozomu) přiřadit tzv. fintess hodnotu určující míru shody kandidátního řešení a specifikace.

Návrh kombinačních obvodů

Pro kombinační obvody to pak znamená, že fitness funkce se provede tak, že se vyhodnotí funkčnost zapojení obvodu postupným předkládáním všech možných vstupních kombinací. Fitness hodnota určitého kandidátního řešení v zásadě odpovídá buď počtu chybných (v případě minimalizace), nebo počtu správných (v případě maximalizace) odezev. Počet vstupních kombinací závisí na řešené úloze.

V návrhu kombinačních obvodů a úplné definice musí být specifikováno 2^k vstupních vektorů, kde k je počet vstupů kombinačního obvodu. Pro každou vstupní kombinaci a dané propojení jednotlivých elementů se vypočítá hodnota na výstupu. Tzn. simuluje se funkce předloženého kombinačního obvodu a porovnává se výstupní kombinace z dané pravdivostní tabulky s odsimulovanou výstupní kombinací. Hodnota fitness pak odpovídá počtu shod s předem zadanými výstupními hodnotami pro všechny vstupní kombinace. Pokud máme např. $k = 4$ vstupů a $l = 2$ výstupů, pak počet vstupních kombinací je $2^k = 2^4 = 16$. Maximální fitness hodnota tedy může být $l * 2^k = 2 * 16 = 32$.

Symbolická regrese

Problém symbolické regrese spočívá v nalezení takové matematické funkce, která co nejlépe aproximuje hledanou funkci f zadanou pomocí množiny vstupích hodnot in a očekávaných výstupních hodnot out. Jednou z možných implementací je, že se určí odchylka od správného řešení. Fitness hodnota se pak vypočte pomocí vzorce

$$\sum_{i=0}^{m} |f(in_i) - out_i|$$

kde m označuje počet vstupních kombinací. Mimo jiné můžeme použít i sumu kvadratických odchylek.

Klasifikace

Pomocí CGP lze řešit problém klasifikace. To znamená, že se bude hledat taková funkce, která rozdělí data do tříd. Vstupními daty jsou jednotlivé dimenze daného vektoru dat. Výstupem pak je informace o tom, zda patří do dané třídy či nikoli. Samotná aproximační funkce f může obsahovat práh, který zajišťuje příslušnost do třídy, kde funkce g simuluje chromozom.

$$f(in) = \begin{cases} +1 & \text{pro } g(in) \geq 0 \\ -1 & \text{pro } g(in) < 0 \end{cases}$$

Funkce funkční množiny Γ obsahují aritmetické operace ($+$, $*$, atd.), relační operátory ($>$, $==$, atp.) a logické operace(and, if, atd.) . Výsledek relačních operátorů většinou bývá 1, nebo 0 datového typu integer, kdežto aritmetické operace jsou datového typu float. Tudíž je nutné jejich hodnoty přetypovávat.

Chceme-li klasifikovat určitá data do více jak dvou tříd, je nutné vygenerovat tolik funkcí kolik je tříd [8].

Další implementace fitness funkce

V případě evolučního umění může být fitness funkce řešena manuálně. Člověk si subjektivně vybere nejlepší obrázek, tím pádem mu přiřadí nejlepší skóre. V jiných případech může být naopak velmi složitá [7].

2.2.4 Optimalizace

Po dosažení maxima fitness – po nalezení správného řešení, se nalezený jedinec může dále optimalizovat. Nejčastěji se redukuje počet použitých uzlů n_{used} a to z důvodu snížení nákladů na tvorbu obvodu. Pro tuto optimalizaci se používá modifikovaná fitness funkce. Její vzorec v případě maximalizace fitness hodnoty je následující:

$$fit = fit_{max} + n_r * n_c - n_{used}$$

Z pohledu návrhu hardware je však někdy výhodnější optimalizovat obvod na zpoždění hradel. Poněvadž řešení s celkovým zpožděním hradel $10T$ s počtem uzlů 42 je rychlejší, než řešení se zpožděním $12T$ a s počtem uzlů 39 [10].

Kapitola 3

Python a možnosti akcelerace výpočtu

Jazyk Python [9] patří do skupiny tzv. interpretovaných jazyků. Výhodou těchto jazyků je přenositelnost kódu. Přenositelnost je zajištěna pomocí tzv. bytecode (portable code). Další výhodou tohoto jazyka je jeho snadná syntaxe, dynamické typování a snadná kompilace programu za jeho běhu. Na druhou stranu typicky trpí v porovnání s jazyky kompilovanými (například jazyk C).

V dnešní době existují dvě vývojové větve jazyka Python. Tudíž záleží také na interpretu, na kterém se bude daná aplikace spouštět. Poněvadž v některých problémech je Python2 rychlejší než Python3. V sekci 3.3 se zmiňuji o PyPy. Jedná se o implementaci interpretu Pythonu pomocí níž můžeme získat zajímavé výsledky.

Pro psaní modulů v Pythonu můžeme využít rozšíření označované jako jazyk Cython. Tento jazyk disponuje statickým typováním proměnných a voláním funkcí jazyka C.

3.1 Tvorba programu v Pythonu

Python pod svou jednoduchostí obsahuje místa, na nichž je snadné udělat chybu. Tím je myšleno, že vývoj v Pythonu je velmi rychlý, ale leckdy se to negativně projeví na rychlosti výsledného programu. V této podkapitole se věnuji zásadám jak psát efektivní kód v Pythonu.

Je lepší odchytávat výjimky než jim předcházet. Tento přístup je nazýván EAFP – Easier to Ask Forgiveness than Permission. Je snadnější žádat o odpuštění nežli o povolení. Tato technika dělá zdrojový kód přehlednějším [6]. Testování objektů příkazem `if` na identitu (`if object is None`) je efektivnější než na rovnost (`if object == None`). Při testování na identitu se porovnávají pouze adresy objektů. Při testování na rovnost se porovnávají i jejich data [6]. Spojování řetězců by mělo probíhat pomocí metody `join()`, neboť tato metoda nabývá lineání složitost. Operátor `+` nabývá složitost kvadratickou a také dochází k typové kontrole objektu [3]. V Pythonu2 je vhodnější použít funkci `xrange()` než `range()`, protože používá generátor. Funkce `range()` vytváří pole, kterým se pak iteruje a zbytečně zabírá mnoho paměti. V Pythonu3 funkce `range()` již používá generátor a funkce `xrange()` byla odstraněna [4]. Volání funkcí v Pythonu (tak jako i v jiných jazycích) je relativně pomalé. U funkcí, jenž se volají ve smyčkách, se vyplatí jejich rozbalení [6]. Rovněž toto platí o časté alokaci paměti ve smyčkách.

3.2 Běžné metody akcelerace výpočtu fitness funkce

Analýzou časové náročnosti jednotlivých kroků evolučního algoritmu můžeme snadno zjistit, že nejvíce času stráví aplikace výpočtem fitness hodnoty. Pro každé kandidátní řešení a každý vektor trénovacích dat je zapotřebí simulovat funkci genotypu.

3.2.1 Paralelní simulace

V případě kombinačních obvodů roste doba výpočtu exponenciálně s rostoucím počtem vstupů. Při naivní simulaci, kdy by se testovaly jednotlivé vstupy tak jak jsou, by bylo nutné ohodnotit 2^k řešení. V případě kombinačních obvodů můžeme využít a využít tzv. paralelní simulaci, kdy v proměnné typu integer může být uloženo 32 bitů. Tudíž se simuluje 32 výpočtů fitness funkce jedním průchodem fitness funkce. V 64 bitové architektuře se pak může simulovat 64 výpočtů jedním průchodem.

Python je však jazyk, který má dynamickou velikost celých čísel (jsou to objekty s názvem `int`). Pro něj není problém provádět 42 bitové operace na 32 bitovém stroji. Z tohoto důvodu může být simulován libovolný počet výpočtů fitness funkce jedním průchodem. Tato vlastnost má však jednu velkou nevýhodu. Tento dynamický integer je alokován jako pole a při operacích (bitový posun, sčítání, odčítání. . .) dochází k jisté režii.

K akceleraci je možné využít SIMD koprocesory, který dovoluje pracovat až s šestnácti 8 bitovými čísly nebo s čtyřmi FP čísly (čísly s pohyblivou desetinnou čárkou) současně. Nevýhodou je však omezená instrukční sada.

Výpočet správně určených bitů

Při výpočtu hodnoty fitness u kombinačních obvodů probíhá mimo jiné výpočet správně určených bitů. Ten můžeme rovněž paralelizovat a to pomocí Hammingovy vzdálenosti. Paralelizace může být tedy provedena aplikací vzorce

$$fitness = zerocount(a \oplus b),$$

kde a je výsledek simulace a b je očekávaný výsledek. Exklusivní součin $\oplus$ způsobí to, že pokud jsou dané bity ekvivalentní, výsledek je nula. Chceme-li tyto bity spočítat, pak aplikujeme funkci $zeroucount()$, která může být implementována různými způsoby.

Před spuštěním CGP se inicializuje vyhledávací tabulka (LUT z angl. Lookup table). Tato tabulka obsahuje hodnoty jedničkových bitů pro daný klíč. Klíč má omezenou délku (8 bitů). Je zapotřebí provést bitové operace tak, aby se pomocí LUT vypočítala Hammingova vzdálenost pro celý registr, který má velikost 64 bitů (popřípadě 32).

Lepší metoda je spočítat dané bity paralelně pomocí bitových operací. Viz algoritmus 2. Tato metoda využívá pouze 12 operací a není tak paměťově náročná jako lookup tabulka. Tudíž naprostou výhodou je to, že je větší pravděpodobnost, že daná tabulka bude v rychlé vyrovnávací paměti (cache) [2].

Algoritmus 2: *Population count*

```
v = v˜- ((v˜>> 1) & 0x55555555);
v = (v˜& 0x33333333) + ((v˜>> 2) & 0x33333333);
c = ((v˜+ (v˜>> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

Využití SIMD koprocesorů

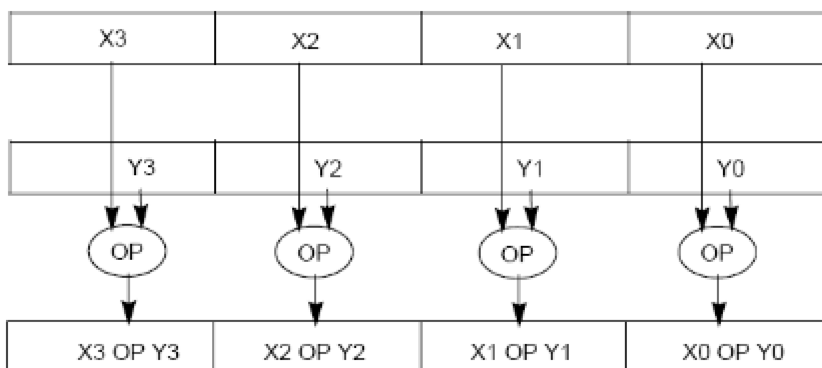
V dnešní době běžný procesor disponuje jednotkou SSE – Streaming SIMD (Single instruction, multiple data) Extension. Jedná se o rozšíření, které přidává procesoru osm 128 bitových registrů. Tento registr může obsahovat buď dvě 64 bitové, nebo čtyři 32 bitové hodnoty čísel v plovoucí řádové čárce. Nad těmito čísly se pak mohou provádět paralelní operace. Toto je zobrazeno na obrázek 3.1. To má za následek to, že pomocí jednotky SSE mohou být simulovány čtyři aritmetické výpočty hodnoty fitness.

Jednotka SSE slouží převážně pro práci s reálnými čísly. Ovšem disponuje i bitovými (logickými) operacemi, jakými je bitový součin (`ANDPD`), bitový součet (`ORPD`), exklusivní součet (`XORPD`) a negovaný bitový součin (`ANDNPD`). Realizace bitové negace se provede pomocí instrukce `ANDNPD`, která se provede nad jedním registrem: `ANDPD xmm1, xmm1`. V případě kombinačních obvodů pak může být simulováno 128 výpočtů jedním průchodem fitness funkce.

V instrukční sadě SSE4.2 (a novější) se nachází instrukce `POPCNT`. Tato instrukce spočítá v registru počet bitů s hodnotou jedna. Princip výpočtu je podobný algoritmu 2.

Technologie Advanced Vector Extensions (AVX) rozšiřuje původní 128 bitové registry jednotky SSE na 256 bitů. Tento 256bitový blok může obsahovat čtyři 64 bitové hodnoty, nebo osm 32 bitových hodnot.

Instrukce SSE či AVX se používají převážně v jazycích symbolických instrukcí (assembler), či v strukturovaných jazycích (jazyk C). Využití těchto koprocesorů v Pythonu je docela problém, neboť se jedná o skriptovací jazyk.



Obrázek 3.1: Provedení SIMD operace

Využití grafického procesoru

Pro paralelní výpočet fitness hodnoty se může využít také grafického procesoru (GPU). Vzhledem k značnému počtu výpočetních jader umožňuje větší míru paralelizace než SIMD koprocesory v procesoru (CPU). Nevýhodou současných GPU je značná režie v důsledku paměťových operací. Tzn. bude-li výpočet fitness funkce probíhat na GPU a algoritmus CGP bude na CPU, pak značnou nevýhodou je přenos dat z operační paměti RAM do paměti grafického procesoru. Tato metoda by se vyplatila pouze pro velký objem dat – řádově MB.

Předchozí problém může být vyřešen tím, že celý algoritmus bude běžet na GPU. Tento přístup však přináší problém. Každé větvení na GPU zastaví všechny vlákna v každém bloku a každý nezarovnaný přístup do paměti způsobí penalizaci.

3.2.2 Akcelerace využívající znalosti fenotypu

V případě, že budeme zohledňovat fenotyp kódovaný pomocí CGP, můžeme výpočet fitness hodnoty značně urychlit.

Před evaluací fintess funkce se vytvoří tabulka použitých uzlů grafu. Následně se prochází daný chromozom odzadu tak, aby se získaly pouze použité uzly. Ty se pak při výpočtu fintess ohodnocují.

Toto řešení však přináší jistou režii. Nehodí se tehdy, pokud fenotyp obsahuje velké množství uzlů grafu.

3.2.3 Kompilace za běhu programu

Většina procesorů v dnešní době zpracovává instrukce zřetězeně tzv. pipelining. Základní myšlenka vychází ze skutečnosti, že zpracování každé instrukce procesorem lze rozdělit do několika fází:

- **Fetch** – Načtení instrukce z paměti.
- **Decode** – Dekódování instrukce. Určí se typ operace, načtou se operandy, se kterými daná instrukce pracuje.
- **Execution** – Provedení instrukce.
- **Write back** – Uložení výsledků zpracované instrukce

Při klasickém zpracování se tak instrukce zpracuje v prvních čtyřech taktech. V dalších čtyřech taktech se zpracuje další instrukce. Při zřetězeném zpracování se instrukce zpracuje v prvních čtyřech taktech, ale další instrukce jsou už rozpracovány. Tudíž v každém dalším taktu je zpracována vždy jedna další instrukce.

Při zřetězeném zpracování dochází problém, když některá z instrukcí je instrukce skoku. Pak je nutné provést vyprázdnění fronty (pipeline flush). Předzpracované instrukce totiž nebudou prováděny, neboť program bude pokračovat tam, kde byl uskutečněn daný skok.

Problémem výpočtu fitness funkce je ten, že obsahuje hodně instrukcí skoku. Ve fitness funkci se nachází smyčka `for()` spolu s dlouhým podmíněným příkazem `if()`. Viz algoritmus 3. V případě symbolické regrese máme k vstupních kombinací. Nepoužíváme-li paralelismus výpočtu, pak je nutné k výpočtu fitness hodnoty provést právě k evaluací. Tolikrát se nám zvýší i počet instrukcí skoku.

Tento problém může být eliminován zkompilováním daného chromozomu za běhu programu a jeho následným spuštěním. Kompilace chromozomu probíhá pouze jednou a to před skutečným výpočtem fitness funkce a snaží se vytvořit co nejefektivnější kód. Tento kód se pak spouští místo původní evaluace [12].

V Pythonu však instrukce skoku neeliminujeme, protože jeho bytecode je stále interpretovaný. Ovšem snížíme počet interpretovaných operací. Relační operátory v podmínkách v Pythonu kontrolují kompatibilitu mezi datovými typy. To bude mít za následek to, že evaluace fitness funkce bude probíhat rychleji.

Kompilace pomocí funkce `compile()`

Překlad za běhu programu můžeme v Pythonu realizovat pomocí funkce `compile()`. Parametry této funkce jsou: daný kód ve formě řetězce; název souboru, do kterého se bude kompilovat; použití kompilovaného kódu (řetězec s hodnotou `exec` či `eval`). Tato funkce

Algoritmus 3: *Výpočet fitness funkce s jejich maximální aritou 2*

Input: $chromFunction$ – pole funkcí, které představuje daný uzel i
a – pole propojení uzlů grafu na jejich první vstupní port
b – pole propojení uzlů grafu na jejich druhý vstupní port.
Output: Pole vypočtených hodnot out

$idx = n_i$
for $i = 0$ to $n_c * n_r$ **do**
  opearce $= chromFunction[i]$
  if operace =='+' **then**
    $out[idx] = out[a[i]] + out[b[i]]$
  else if operace =='−' **then**
    $out[idx] = out[a[i]] - out[b[i]]$
  else if operace =='∗' **then**
    $out[idx] = out[a[i]] * out[b[i]]$
  else if operace =='sin' **then**
    $out[idx] = sin(out[a[i]])$
    ...
  end if
  $idx++$
end for

nám vrací zkompilovaný bytecode. Ten se následně spouští jako parametr funkce `exec()`, nebo `eval()`. Záleží na tom jak byl daný kód zkompilován. Kód, který je spouštěn pomocí funkce `exec()`, může obsahovat podmínky, cykly či funkce. Naopak kód, který je spouštěn pomocí funkce `eval()`, slouží výhradně pro aritmetické funkce.

Nevýhodou funkce `compile()` je, že má velkou režii. Důvodem toho je fakt, že kromě kompilace kódu provádí syntaktickou kontrolu.

Kompilace do bytecode

Pro vytvoření vlastního bytecode můžeme využít funkci `CodeType()` modulu `types` ze standardní knihovny Pythonu. Tato funkce přijímá 13 povinných parametrů v Pythonu3. (V Pythonu2 je jich 12.) Jejich podrobný popis naleznete v apendixu, nebo v dokumentaci Pythonu. Mezi nejdůležitější parametry patří: velikost zásobníku, lokální proměnné, konstanty kódu a samotný bytecode. Velikost zásobníku nám udává, jak velký bude zásobník interpretu Pythonu. Lokální proměnné a konstanty jsou uloženy jako typ `tuple()`. Samotný bytecode je pak uložen jako pole znaků, respektive bajtů [9].

Funkce `compile()` vrací objekt `Code`. Jejím nejdůležitějším atributem je `co_code`. Ten nám representuje daný bytecode. Příklad bytecode:

```
>>> c = compile("y = x/0","","exec");
>>> print(c.co_code)
0x6500006400001b5a010064010053
```

Tento bytecode pak může být dekompilován pomocí funkce `dis()` modulu `dis` [9].

```
1           0 LOAD_NAME                0 (x)
            3 LOAD_CONST               0 (0)
            6 BINARY_TRUE_DIVIDE
            7 STORE_NAME               1 (y)
           10 LOAD_CONST               1 (None)
           13 RETURN_VALUE
```

Nejvíce vlevo sloupec značí číslo řádku překládaného kódu, v našem případě to je 1. Druhý sloupec čísel (0, 3, 6, 7...) je index bytecode. Následuje sloupec se jmény instrukcí odpovídající danému indexu do bytecode. Sémantika těchto instrukcí je dostupná v dokumentaci [9]. Některé z nich (v našem případě `STORE_NAME`, `LOAD_NAME` a `LOAD_CONST`) potřebují argumenty. Ty jsou uvedeny v předposledním sloupci. Ten udává index do datové struktury `tuple`, která buď obsahuje konstanty, nebo proměnné daného kódu. Jejich hodnoty či jejich názvy jsou interpretovány v posledním sloupci.

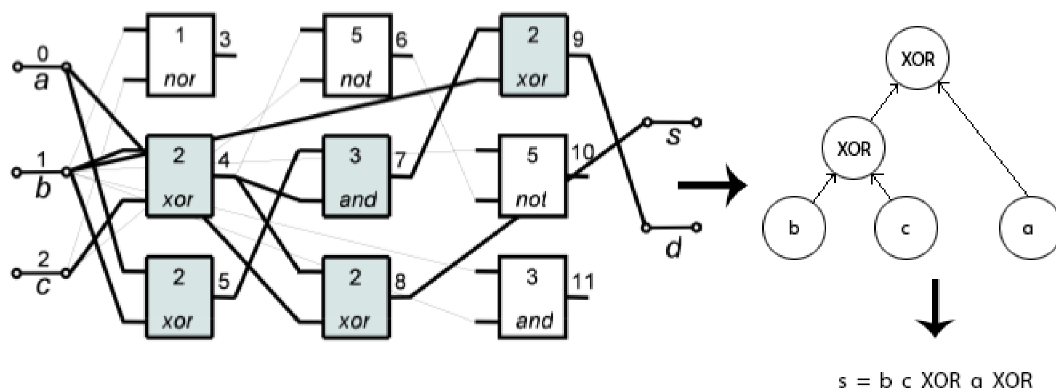
Číselné hodnoty instrukcí jsou uvedeny v souboru `$PYTHON_LIB_DIR/opcode.py`. Mimo jiné mohou být získány přímo v interpretu Pythonu a to takovým způsobem, že použijeme hodnoty druhého sloupce (0, 3, 6, 7...) jako index do daného bytecode. Tedy například, chceme-li zjistit jaké číslo má instrukce představující dělení – `BINARY_TRUE_DIVIDE`, pak použijeme tento příkaz `print(kod.co_code[6])`.

Bytecode by měl být ukončen posloupností příkazů `LOAD_CONST (None)`, `RETURN_VALUE`.

Výsledné použití v kartézském genetickém programování by znamenalo, že by se pro každý uzel grafu vygeneroval stejně dlouhý kód. Ten by prováděl danou operaci. Tento kód by byl uložen v poli, které by bylo velké $l_c = n_r * n_c * l_n + 4$, kde n_r je počet řádků grafu, n_c počet sloupců grafu a l_n udává velikost bytecode pro jeden uzel. Zpravidla je to maximální délka kódu, kterou uzel může dosáhnout. Dále je potřeba připočíst konstantu 4, která představuje zakončení bytecode.

Kompilace by se prováděla tak, že díky konstantní délce bytecode uzlu se přepíše funkce, kterou daný uzel vykonává a jeho propojení. Tím se překlad bytecode urychlí. Hlavně díky tomu, že neprobíhá žádná syntaktická analýza.

Problém u této metody nastane tehdy, pokud chceme evaluovat pouze fenotyp daného chromozomu, neboť kód by pak měl proměnnou délku. Toto řeším v kapitole 4.1.3.



Obrázek 3.2: Kompilace chromozomu do postfixové notace

Kompilace bytecode do postfixové notace

Princip interpretu Pythonu je založen na zásobníkové architektuře. Podobně je tomu také u JVM (Java Virtual Machine). Operace v této architektuře se provádí většinou nad vrcholem zásobníku. Toto může být využito tak, že se daný chromozom zkompiluje do postfixové notace (do reverzní polské notace). Fenotyp chromozomu se pro jeden určitý výstup podobá datové struktuře strom. Tímto stromem pak můžeme procházet takovým způsobem, že z něj vznikne postfixový kód, který je vhodný pro interpret Pythonu. Tento převod pak zobrazuje obrázek 3.2.

Může nastat situace, kdy na výstup daného uzlu jsou napojeny víc jak dva uzly. Tím pádem dojde k opětovnému vypočtu stejné hodnoty daného uzlu. Tento nedostatek být vyřešeno tím, že se v bytekódu tato hodnota uloží do pomocného pole. K opětovnému výpočtu daného uzlu nedojde, načte se uložená hodnota.

Tato kompilace má sice mnohem větší režii než předchozí, ale díky ní můžeme vytvořit efektivnější kód.

3.3 Využití PyPy pro akceleraci programu v Pythonu

„If the implementation is hard to explain, it's bad idea, except PyPy."

(Benjamin Peterson, hlavní vývojář PyPy)

PyPy je interpret Pythonu, který se specializuje na rychlost výsledného programu. První verze interpretu PyPy byly napsány čistě v Pythonu, odtud vychází název. PyPy funguje na principu Just-in-time kompilace kódu, rozbaluje smyčky a tím optimalizuje kód [1].

Problém nastane tehdy, pokud chceme spustit svůj kompilovaný kód v smyčce. Potom musíme jeho spuštění zabalit do funkce. Jinak dostaneme neefektivní kód. Rychlost tohoto kódu můžete porovnat v tabulce 3.1.

`exec()` v cyklu `for()`	`exec()` ve funkci `myexec()` a volání `myexec()` v cyklu `for()`
1492	4357

Tabulka 3.1: Počet evaluací za sekundu spouštěného bytecode v PyPy. Testováno na evaluaci 4 + 4 sčítačky

3.4 Cython

Cython je generátor ze syntaxe Pythonu rozšířeného o datové typy do jazyka C. Mimo jiné podporuje volání funkcí jazyka C. Díky tomu můžeme napsat efektivní programy. Jazyk Cython nám umožňuje:

- napsat kód v Pythonu a z něj získat jeho příslušný kód v jazyce C nebo C++.
- jednoduše zefektivnit kód Pythonu tak, že použijeme statické typování proměnných.
- využívat i dynamické typování Pythonu.
- vytvořit efektivní modul, který pak může být spouštěn v Pythonu.
- používat knihovny a moduly jazyka C či C++

Pomocí Cythonu může být vytvořen modul takovým způsobem, že z kódu napsaného v Cythonu se nejdříve vygeneruje kód v jazyce C. Ten je pak přeložen pomocí překladače jazyka C (např. GCC, G++). Tím vznikne knihovna, která pak může být použita v interpretu Pythonu.

3.4.1 Syntaxe Cythonu

Statické typování

Cython disponuje statickým typováním. Toho se docílí tím, že před danou funkci nebo proměnnou se připíše `cdef` spolu s jejím typem, například: `cdef int 42`. Klíčové slovo `cdef` udává, že se jedná o typ jazyka C. Interpret Pythonu nedovoluje volat funkce deklarované tímto způsobem. Musí být deklarovány buď klasicky pomocí `def`, nebo pomocí `cpdef`. Funkce deklarované `cpdef` mohou být přetěžovány pomocí `def` funkcemi, nebo samy mohou přetěžovat `cdef` funkce.

Hlavní rozdíly mezi výrazy jazyka C a Cython

Tento jazyk má v některých případech mírně odlišnou syntaxi a sémantiku od jazyka C. To je zapříčiněno kvůli Pythonu a jeho syntaxi a sémantice.

- V Cythonu neexistuje operátor `->`. Namísto `p->a` se použije `p.a`.
- V Cythonu neexistuje unární operátor dereference `*`. Namísto `*p` se použije `p[0]`
- V Cythonu můžeme použít oeprátor reference `&` se stejnou sémantikou jako je v jazyce C.
- Přetypovat proměnné můžeme pomocí `<typ>prom`, například:
  ```
  cdef int * a, float * b
  a = <int *> b
  ```

Volání knihovních funkcí jazyka C

Chceme-li využít určitou funkci ze standardní knihovny jazyka C, musíme nejdříve naimportovat danou knihovnu pomocí příkazu `cdef extern from "nazev_knihovny":`. Dále musíme uvést v jeho bloku (jmenném prostoru) jméno každé funkce, kterou chceme z daného modulu použít, společně s jejími parametry a její návratovou hodnotou. Tyto příkazy musí být zadány před voláním dané funkce.

Příklad deklarace funkce `malloc()`.
```
cdef extern from "stdlib.h":
    void * malloc(int a)
```

3.4.2 Just-in-time kompilace

Jazyk Cython může být využit jako Just-in-time kompilátor binárního kódu. Tento kód by se kompiloval za běhu skriptu v Pythonu. Vznikl by efektivní kód. Bohužel kompilací každého chromozomu by předcházel překlad Cythonu a jazyka C. Vyplatilo by se to pouze pro velké množství dat.

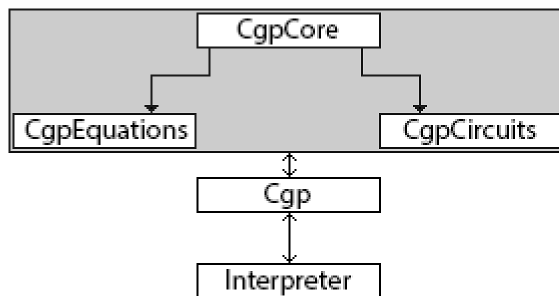
Kapitola 4

Implementace navrženého řešení

Implementace řešení probíhala v několika etapách. Nejdříve se naprogramoval algoritmus čistě pro návrh kombinačních obvodů. V průběhu optimalizace kombinačních obvodů pomocí Just-in-time kompilace byla přidána symbolická regrese, neboť u ní můžeme spatřit lepší výsledky. Viz kapitola 5. Následně se použil překlad chromozomu do postfixové notace bytecode. Celý průběh etapy vývoje byl brán ohled na kompatibilitu mezi interprety Python2, Python3 a PyPy. Jako výsledek této práce byl vytvořen modul CGP, který je napsán v Cythonu. Ten se pak může spouštět přímo v interpretu v Pythonu.

4.1 CGP v Pythonu

Výsledný program v Pythonu lze rozdělit do několika funkčních bloků. Jádro algoritmu je uloženo v souboru `cgp_core.py`. Tento algoritmus vychází z koncepce uvedené v kapitole 2. V souborech `cgp_circuit.py` a `cgp_equation.py` se pak CGP specializuje na specifický problém. Tyto soubory se pak v průběhu aplikace akceleračních technik měnily nejvíce. Modul `cgp.py` slouží především jako komunikační prostředek mezi algoritmem CGP a interpretem Pythonu.



Obrázek 4.1: Diagram návrhu komunikace mezi moduly CGP

Jádro algoritmu

Jádro algoritmu CGP (třída `CgpCore`) je k dispozici v souboru `cgp_core.py`. Funkce `runAscending()` spouští CGP. Rodič se vybírá na základě maximalizace fitness hodnoty, jenž se v průběhu evoluce zvyšuje. Toto spouštění algoritmu slouží především pro kombinační obvody.

Funkce `runDescending()` rovněž spouští CGP. Výběr kandidátního řešení se určuje na základě minimalizace hodnoty fitness, která se v průběhu evoluce snižuje. Tato funkce se používá pro řešení problému symbolické regrese.

Funkce `_initCGPRun()` slouží pro inicializaci běhu CGP. Nastavuje příslušné hodnoty jakou je například velikost chromozomu, ukazatel na poslední gen chromozomu, celkový počet uzlů grafu atd. Funkce `_initLevelBack()` inicializuje pole hodnot, které určují míru propojení jednotlivých sloupců. Tyto hodnoty jsou pak využity v následné mutaci `__mutation()`, aby nedošlo k chybnému propojení grafu. Funkce `_initPopulation()` inicializuje populaci, která je vygenerována pseudonáhodně. Funkce `initOutputBuff()` vytváří pomocné pole na ohodnocení jedince. Další funkce `_initUsedNodes()` inicializuje pomocné pole pro evaluaci fenotypu daného řešení. Tento fenotyp se pak nalezne pomocí funkce `_usedNodes()`.

Dále se v tomto modulu nachází dvě pomocné funkce pro ladění `resultChrom()` a `showChrom()`, které zobrazují chromozom.

Komunikační modul

Komunikační modul slouží ke komunikaci interpretu s algoritmem CGP. Obsahuje třídu `Cgp`, jejíž konstruktor inicializuje implicitní hodnoty. Funkce `file()` a `data()` nastavují vstupní a výstupní data CGP. Další funkce `graph()` mění rozměry grafu a jeho propojení. Metoda `run()` spustí algoritmus CGP. Podle zadaných dat se pak vybere, jestli se jedná o symbolickou regresi, či o návrh kombinačních obvodů. Vstupní parametry těchto funkcí jsou popsány v kapitole D.

Metody `showChrom()` a `resultChrom()` zobrazují dané chromozomy. Pomocí zbylých metod[1] (`allLogicalOperations()`, `reedMuller()`, `symbolicRegression()`) se nastaví funkční množina. Tvorba vlastní funkční množiny záleží na použité akcelerační technice. Atributy `functionTable` a `functionTableOp` jsou seznamy řetězců představující název dané funkce. Atribut `functionArity` je rovněž reprezentován seznamem, kde jeho jednotlivé položky značí aritu dané funkce. Maximální arita funkcí je omezená na 2.

Specializace CGP

V souborech `cgp_circuit.py` a `cgp_equation.py` se nachází implementace tříd `CgpCircuit` a `CgpEquation`. Tato implementace je specifická pro konkrétní řešený problém. V současné době jsou podporovány dva typy aplikací – evoluční návrh kombinačních obvodů a řešení problému symbolické regrese.

V těchto souborech jsou mimo jiné řešeny různé akcelerační techniky, které jsou popsány v následujících podkapitolách.

4.1.1 Naivní řešení evolučního návrhu kombinačních obvodů

První algoritmus evaluace fitness byl implementován čistě pro návrh kombinačních obvodů. Výhodou tohoto přístupu je snazší tvorba trénovacích dat a jednonásobný průchod obvodem

[1] Upozorňuji, že tyto funkce se nevolají přes atribut `function`, tomu se děje pouze u modulu zkompilovaného přes Cython + GCC. Viz kapitola D.

během výpočtu fitness hodnoty. Cílem bylo zjistit výpočetní náročnost řešení založeného na využití dynamického integeru. Viz kapitola 3.2.1.

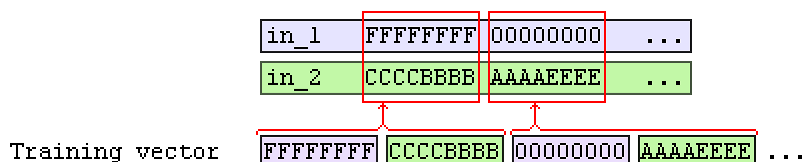
Toto řešení však není vhodné pro následný přepis do Cythonu, neboť integery jazyka C mají velikost 64 bitů (popřípadě 32 bitů). Daná vstupní/výstupní data nejsou rozdělena do trénovacích vektorů. Tudíž by tento výpočet funkce nebyl vhodný pro symbolickou regresi.

Tvorba vlastních funkcí je možná záměnou položek v seznamu funkcí `functionSet`. Jednotlivé položky představují odkaz na danou funkci.

4.1.2 Převod dat do trénovacích vektorů

Druhá implementace se odlišuje od předchozí v reprezentaci trénovacích dat. Trénovací data jsou složena ze sady celočíselných hodnot a jsou reprezentována pomocí běžného datového typu integer. Snahou bylo ověřit, zda-li tento tradiční způsob reprezentace dat poskytuje větší výkonnost.

Data se při inicializaci algoritmu předzpracují do těchto trénovacích vektorů (Funkce `__convertData()`) a to takovým způsobem, aby jednotlivé dimenze byly za sebou. Toto je znázorněno na obrázku 4.2. Takovéto trénovací vektory jsou celkem dva – jeden pro vstup a druhý pro výstup. Pomocí vstupních trénovacích vektorů se pak načítají data, kterými se celá populace ohodnotí pomocí funkce `__evalFitness()`.



Obrázek 4.2: Konvertování dat na trénovací vektory

Tvorba vlastních funkcí je stená jako u předchozí podkapitoly 4.1.1.

4.1.3 Just-in-time kompilace pomocí funkce `compile()`

Jako třetí byla vytvořena implementace využívající Just-in-time kompilaci. Princip této akcelerační metody je uveden v kapitole 3.2.3. Při inicializaci `Cgp` objektu se vytvoří pole, které pak bude obsahovat bytecode. Toto provádí funkce `__initByteCodes()`. Bytecode je pak spouštěn při evaluaci pomocí funkce `exec()`.

Může být použito hned několika metod, které nám dovolí zkompilovat chromozom. Tou nejsnazší je jej zkompilovat pomocí funkce `compile()`. Chromozom však musí být předem přeložen do řetězce. Tomu se tak děje ve funkci `__getCode()`.

K vytváření vlastních funkcí slouží atribut `functionSetStr`. Tato funkce představuje pole řetězců, kde jednotlivý řetězec představuje operaci uzlu vhodnou na následné zkompilování.

4.1.4 Just-in-time kompilace do bytecode

Další implementace spočívá ve vlastní kompilaci chromozomu ve funkci `__getByteCode()`. Princip je ten, že pro každý uzel se vytvoří úsek kódu. V něm se pak mění funkce, kterou

daný uzel představuje. Problém nastává tehdy, chceme-li evaluovat pouze fenotyp daného chromozomu, neboť bytecode musí obsahovat ukončovací 4 byty. Při kompilaci bytecode uzlu se vždy jeho první čtyři byty přepíší na dané hodnoty.

4.1.5 Just-in-time kompilace do bytecode v postfixové notaci

Další uvedenou akcelerační technikou je kompilace bytecode do postfixové notace. Tomu se děje ve funkci `__getByteCodePostfix()`. Postfixová notace se tvoří pomocí rekurzivního volání funkce `__getBc()`. Problém nastává tehdy, vypočítáváme-li některý uzel několikrát. Provede se totiž několik zbytečných výpočtů. Toto je řešeno tím, že při tvorbě fenotypu se určí kolik uzlů je napojeno na daný výstup uzlu. Pokud byl napojen víckrát než jednou, uloží se mezivýsledek do pomocného pole `outputBuff`. Pak v poli použitých uzlů `usedNodes` se nastaví, že je daný výsledek uložen. Pokud je uzel označen za uložený, pak se v bytecode načítá hodnota z pole `outputBuff`.

Tvorba vlastních funkcí v předchozích dvou kapitolách je poněkud složitější, neboť je potřeba znát bytecode. Bytecode ovšem není zdokumentován a je nutné jej získávat ručně. Viz kapitola 3.2.3. Tento bytecode se nachází v atributech `functionBCx`, kde `x` představuje číslo v rozmezí od 1 do 5.

4.2 Cython

Tato podkapitola se zabývá implementovanými technikami využívající Cython.

4.2.1 Tvorba modulu

Vzhledem k tomu, že v Pythonu (bez standardních modulů) máme pouze datový typ slovník a seznam, vzniká při vytváření nových položek režie. Aby se tomu předešlo v případě velkých polí, existuje modul `array`. Jedná se o modul Pythonu napsaný v C, který umí alokovat a přistupovat ke klasickému C poli. Funkcionalita je sice omezená, ale dosáhneme zrychlení.

Manipulace s tímto datovým typem však v Cythonu dělá problém. Jedná se o datový typ vygenerovaný Pythonem a tudíž se pracuje s pamětí přidělenou pro Python. Chceme-li měnit tyto data, kompilátor zahlásí chybu. Proto byly tyto pole vytvořeny pomoci funkce `malloc()` ze standardní knihovny jazyka C.

Další problém s jazykem Cython byl návrh algoritmu. Bylo použito statické typování tříd, které je v Cythonu podobné spíš jazyku C než Pythonu. Původní třídy obsahovaly dynamicky typované atributy, což vedlo k nucenému přepsání celého algoritmu. Upustilo se od původního objektového návrhu, který je znázorněn na obrázku 4.1. Byla vynechána dědičnost modulu `CgpCore`. Tento modul se zcela odstranil, jeho atributy a metody se pak implementovaly v obou specializovaných souborech. Předávání funkční množiny se děje přes objekt, kvůli lepší čitelnosti. Veškeré funkční množiny jsou dostupné v souboru `cgp_functions.pyx`. Dále byla potřeba drobná úprava původního pole trénovacích vektorů (Viz podkapitola 4.1.2). Mezi dva vstupní vektory se vložil k tomu odpovídající výstupní vektor.

4.2.2 Akcelerace modulu

Z důvodu OOP je vhodné mít funkce uloženy v poli. Díky tomu objektově orientovaný přístup nabývá jisté penalizace – časté volání funkcí. Tento nedostatek se odstraní tak, že se funkce nahradí přímým voláním.

Tedy například kód:

```
out[j] = self.functionSet[chrom[idx]](a, b)
```

se rozepíše na:

```
op = chrom[idx]
if   op == 0: c = a + b
elif op == 1: c = a - b
elif ...
```

Kapitola 5

Experimentální vyhodnocení

Tato kapitola obsahuje vyhodnocení výkonnosti jednotlivých navržených implementací, které byly popsány v kapitole 4. Výkonnost bude posuzována z pohledu počtu evaluací, které je schopna daná implementace evolučního algoritmu ohodnotit v rámci jednotky času.

Nejprve bude popsán způsob ověření korektní funkce a pak se budu věnovat jednotlivým vyhodnocením. V podkapitole 5.2 se věnuji vlivu paralelní simulace na výkonnost algoritmu. Porovnání rychlosti naimplementovaných JIT kompilací naleznete v podkapitole 5.3. V obou podkapitolách mimo jiné srovnávám dosažené výsledky interpretů. V další podkapitole 5.4 jsou vyhodnoceny výsledky použitých implementaci v jazyce Cython.

5.1 Verifikace korektnosti fitness funkce

Správnost výpočtu fitness hodnoty byla u každé akcelerační metody ověřena následujícím způsobem. CGP se vždy inicializovalo chromozomem kódujícím korektní řešení. Tento jedinec měl předem vypočtenou fitness hodnotu, která byla sledována.

Požadavkem pro určení správnosti je, aby CGP algoritmus byl inicializován stejně jako vkládaný chromozom. Jinými slovy, aby oba (jak CGP tak i chromozom) měli stejné parametry a to včetně vstup/výstupních dat a včetně uspořádaní funkční množiny. Pro symbolickou regresi byl použit tento chromozom, který představuje výslednou funkci pro logaritmus:

```
[0,0,7, 1,0,4, 1,0,8, 2,0,7, 4,3,5, 5,5,4, 2,3,8, 6,2,8, 8,4,7, 2,4,5, 7,9,0,
 9,8,6, 12,10,6, 13,13,4, 10,0,2, 13]
```

Jeho hodnota fitness je přibližně 123 pro data v souboru `logx.txt`. Funkční množina tohoto chromozomu pak odpovídá této uspořádané funkční množině:

$$\{0.25, 0.50, 1.00, id, add, sub, mul, div, sin\}$$

.

Pro návrh kombinačních obvodů jsem využil chromozom představující $4+4$ sčítačku:

```
[2,6,7,   7,3,5,   9,8,6,   5,1,3,   2,6,7,   2,8,1,   13,4,0,   0,4,7,   8,9,3,
 11,5,2,  4,4,7,   15,17,7, 10,13,6, 2,14,1,  15,3,3,  3,7,3,    13,14,1, 2,4,2,
 25,16,3, 5,26,1,  11,20,5, 28,19,7, 0,4,1,   23,2,0,  25,2,7,   20,11,7, 30,27,0,
 18,29,1, 35,0,4,  10,0,0,  18,16,1, 30,10,6, 28,37,2, 32,31,1,  31,29,5, 28,19,7,
 18,33,1, 29,15,6, 45,34,2, 18,46,1, 47,35,44,38,23]
```

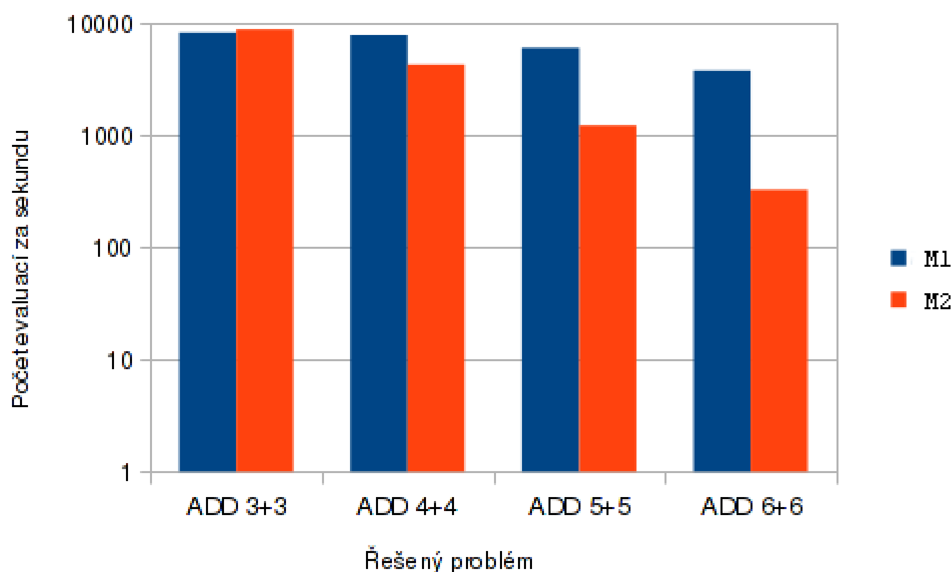
Jeho fitness hodnota odpovídá 1280, počet použitých uzlů je 23. Funkční množina tohoto chromozomu pak odpovídá této uspořádané množině:

$$\{id, and, or, xor, not, nand, nor, nxor\}$$

5.2 Vliv paralelní simulace na výkonnost

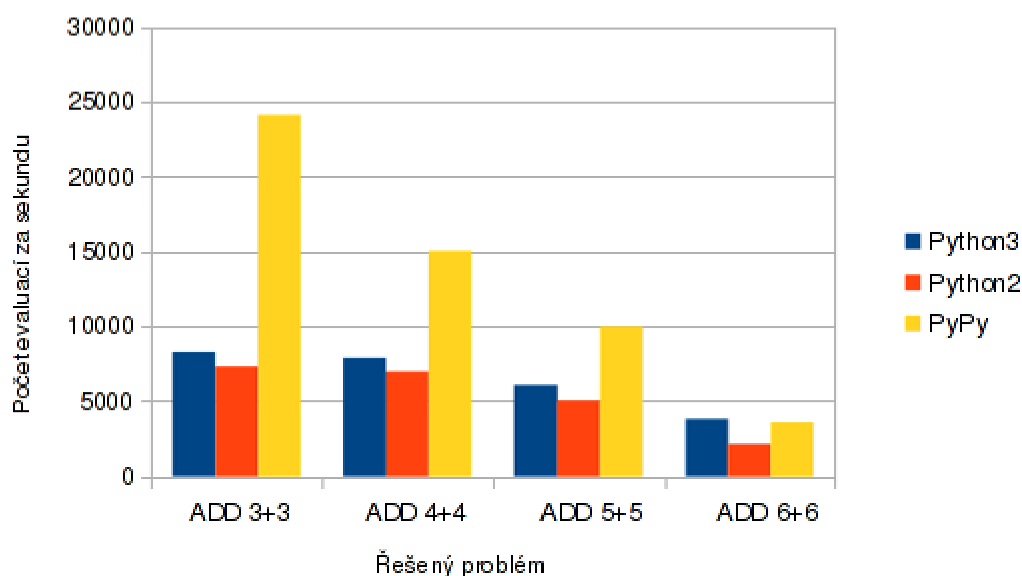
Pomocí paralelního výpočtu jedince bychom procházeli kandidátní řešení pouze jednou. Tudíž v idealním případě platí rovnice $T = t_{CGP} + t_{FIT} * 1$, kde t_{CGP} je čas potřebný pro algoritmus CGP (mutace, selekce), t_{FIT} pak značí výpočet fitness funkce. Ten se provádí pouze *jednou*.

Dynamický integer v Pythonu nám dovoluje simulovat paralelismus výpočtu pro obvody (metoda M1). To má za následek to, že se doba výpočtu fitness funkce t_{FIT} s rostoucím počtem vstupů obvodu zvyšuje – počet evaluací za sekundu se pak snižuje. Tento přístup je však efektivnější, než kdybychom v Pythonu evaluovali na 64 bitovém stroji pouze 64 bitů jedním průchodem (metoda M2), neboť by to znamenalo více operací pro interpret Pythonu. Viz graf 5.1.



Obrázek 5.1: Výkonnost standardní implementace a implementace využívající techniky paralelní simulace na sadě čtyř benchmarkových obvodů v Pythonu3

Dalším vlivem na rychlost paralelní simulace je používaný interpret. Na grafu 5.2 můžete vidět, že nejlépe dopadl PyPy, následován Pythonem3 a jako poslední dopadl Python2. Pokud by se použily náročnější benchmarkové úlohy, dovolím si vyslovit hypotézu, že Python3 by byl dokonce rychlejší.

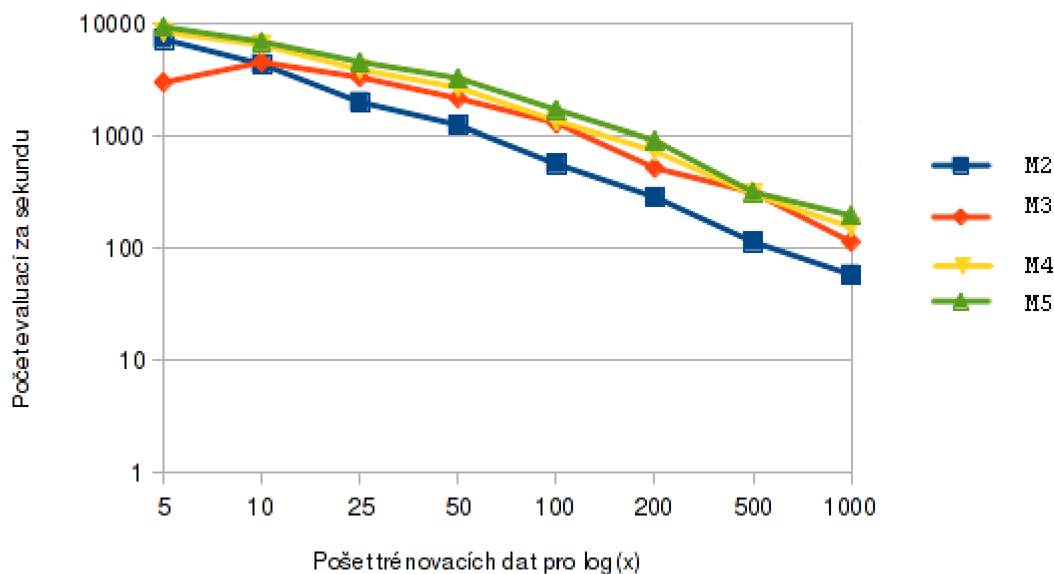


Obrázek 5.2: Výkonnost paralelního výpočtu jednotlivých interpretů na sadě čtyř benchmarkových obvodů

5.3 Vyhodnocení rychlosti implementace využívající kompilaci chromozomu

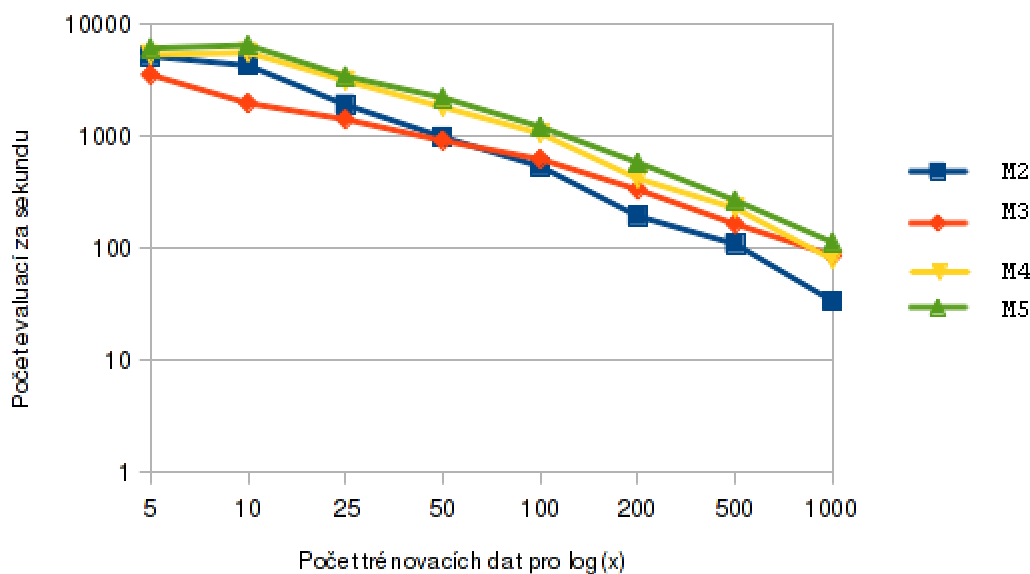
V kapitolách 3.2.3 a 4.1.3 byly představeny možnosti jak lze využít kompilaci k urychlení CGP. Cílem tohoto experimentu je předvést, že záleží na způsobu kompilace kódu. Byly vybrány tři kompilace: pomocí vestavěné funkce `compile()` (metoda M3), vlastní kompilace bytecode (metoda M4) a vlastní kompilace bytecode do postfixové notace (metoda M5).

Z grafu 5.3 lze usoudit, že pro Python2 dopadla kompilace bytecode do postfixové notace nejlépe (metoda M5). Vestavěná funkce `compile()` se v tomto případě vyplatí, máme-li víc jak 25 trénovacích dat.



Obrázek 5.3: Výkonnost použitých JIT kompilací v Pythonu2

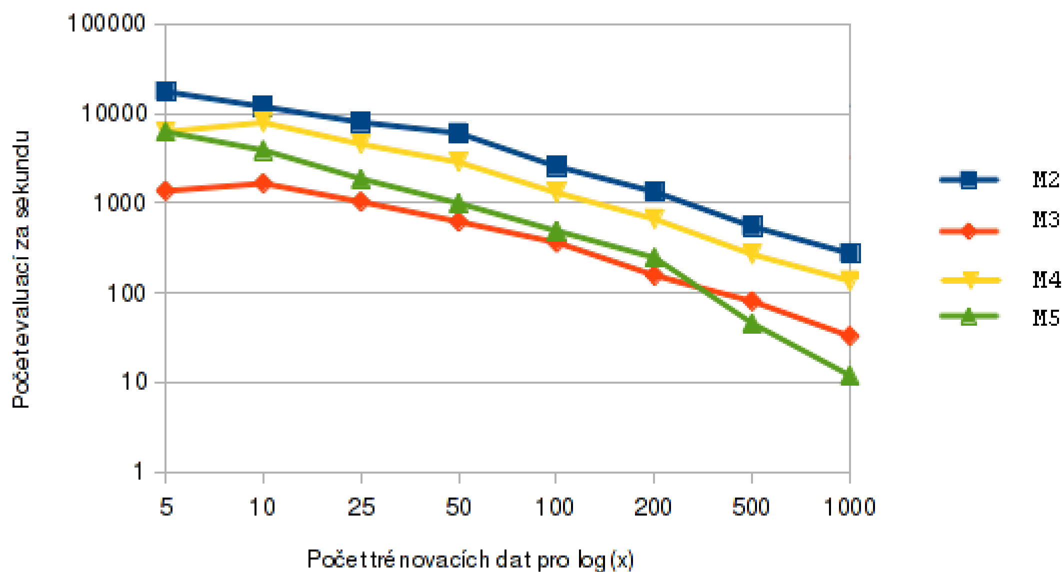
Python3 na rozdíl od Pythonu2 má pomalejší kompilaci kódu pomocí funkce `compile()` (M3). Na grafu 5.4 můžete spatřit, že se takováto kompilace vyplatí tehdy, použije-li se víc jak 100 trénovacích dat. Kompilace bytecode do postfixové notace (M5) rovněž dopadla nejlépe.



Obrázek 5.4: Výkonnost použitých JIT kompilací v Pythonu3

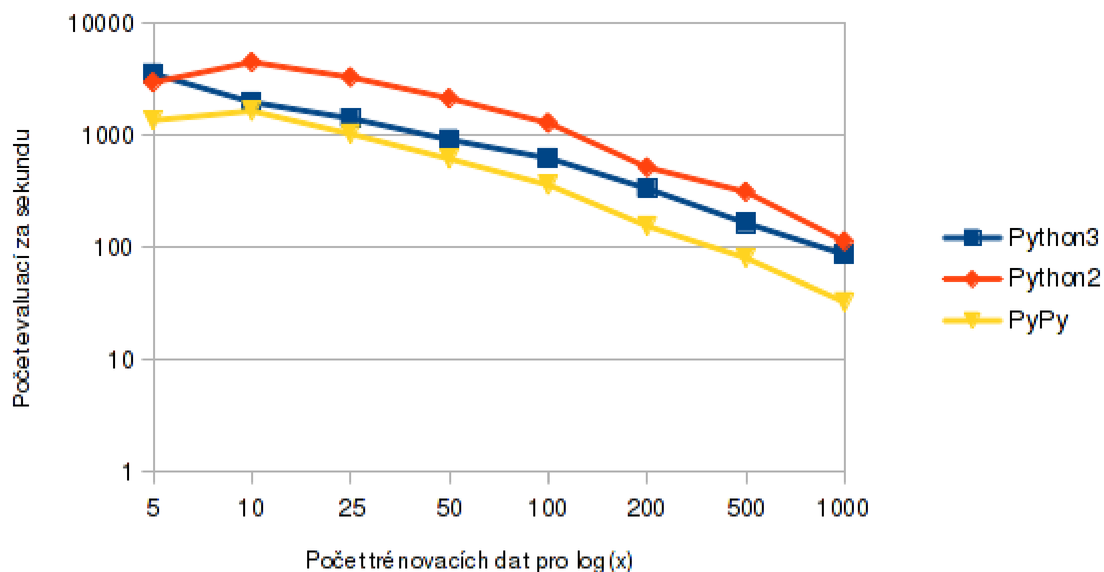
Nejhůře v JIT kompilaci dopadl interpret PyPy. Na grafu 5.5 můžete spatřit, že u něj nedošlo k žádnému urychlení. Důvodem je to, že se sám snaží optimalizovat kód. Rozba-

luje smyčky, redukuje volání funkcí, provádí svou vlastní JIT kompilaci kódu. Toto bylo vysvětleno v kapitole 3.3. Postfixová notace bytecode se vytváří pomocí rekurze. To má za následek to, že tato metoda pro něj dopadla nejhůře.



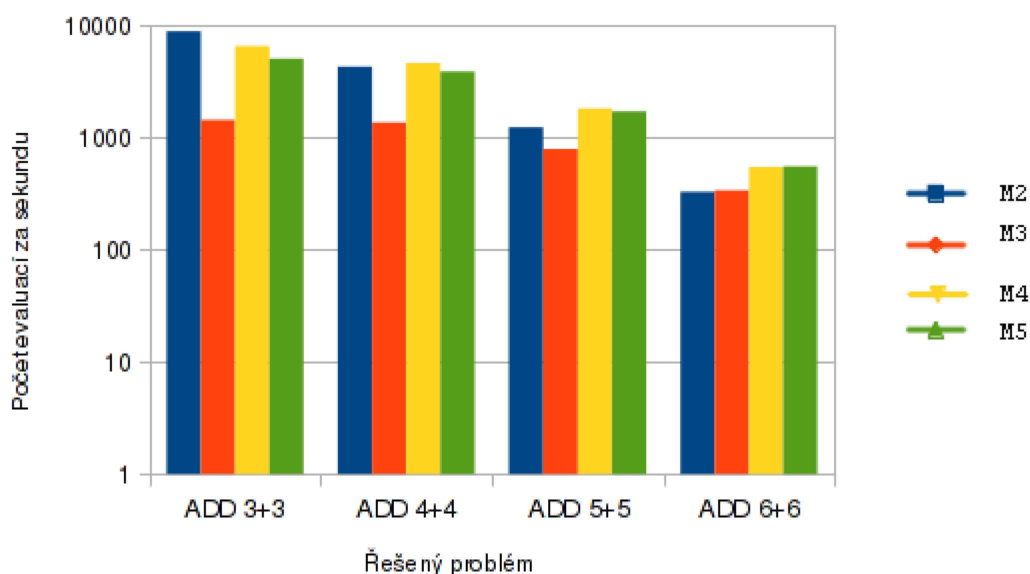
Obrázek 5.5: Výkonnost použitých JIT kompilací v PyPy

Na grafu 5.6 můžeme porovnat rychlost jednotlivých interpretů v kompilaci pomocí funkce `compile()`.



Obrázek 5.6: Porovnání výkonnosti kompilace pomocí funkce `compile()` pro jednotlivé interprety

Srovnáme-li dříve dosažené výsledky s grafem 5.7, zjistíme, že se pro kombinační obvody nevyplatí kompilace kódu do postfixové notace. Tato tvorba kódu má jednak vyšší režii a jednak obvody mají více výstupů než v problému symbolické regrese, kde je výstup zpravidla jeden.



Obrázek 5.7: Porovnání výkonnosti JIT kompilací v Pythonu2 pro kombinační obvody

V tabulce 5.1 můžete porovnat rychlost počtu evaluací za sekundu jednotlivých kompilací za použití Pythonu2. Jako další benchmarkové úlohy byly vybrány funkce $cos(x)$, $sin(x)$, $log(x)$ a $xxyz(x)$. Počet trénovacích vektorů byl vybrán 1000. Z dosažených výsledků lze usoudit, že rychlost algoritmu se zvýšila 3.96 krát.

	cos(x)	sin(x)	log(x)	xxyz(x)
Bez akcelerace (M2)	40	43	58	52
Vlastní kompilace do postfixové notace (M5)	176	188	196	193

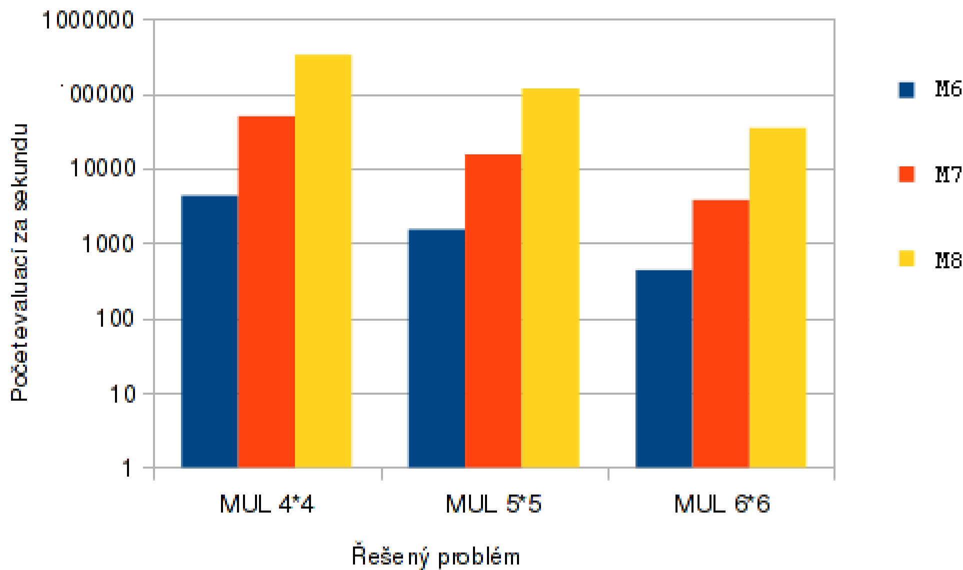
Tabulka 5.1: Počet evaluací za sekundu pro benchmarkové úlohy obsahující 1 000 trénovacích vektorů pro Python2

5.4 Vyhodnocení rychlosti implementace využívající Cython

Byly vytvořeny tři různé implementace, které jsou popsány v kapitole 4.2. První implementací bylo, že ze zdrojového kódu Pythonu se vygeneroval kód do jazyka C. Podle vygenerovaného kódu se pak vytvořil modul (metoda M6). Jazyk Cython nám dovoluje akcelerovat skript pomocí statického typování, jak bylo uvedeno v kapitole 4.2 (metoda M7). Ještě efektivnější řešení dostaneme, rozbalíme-li dané funkce, tak jak bylo popsáno v kapitole 4.2.2 (metoda M8). Toto řešení pak přibližně odpovídá implementaci v jazyce C. Viz tabulka 5.2. Rychlost těchto akceleračních metod můžete spatřit na obrázku 5.8.

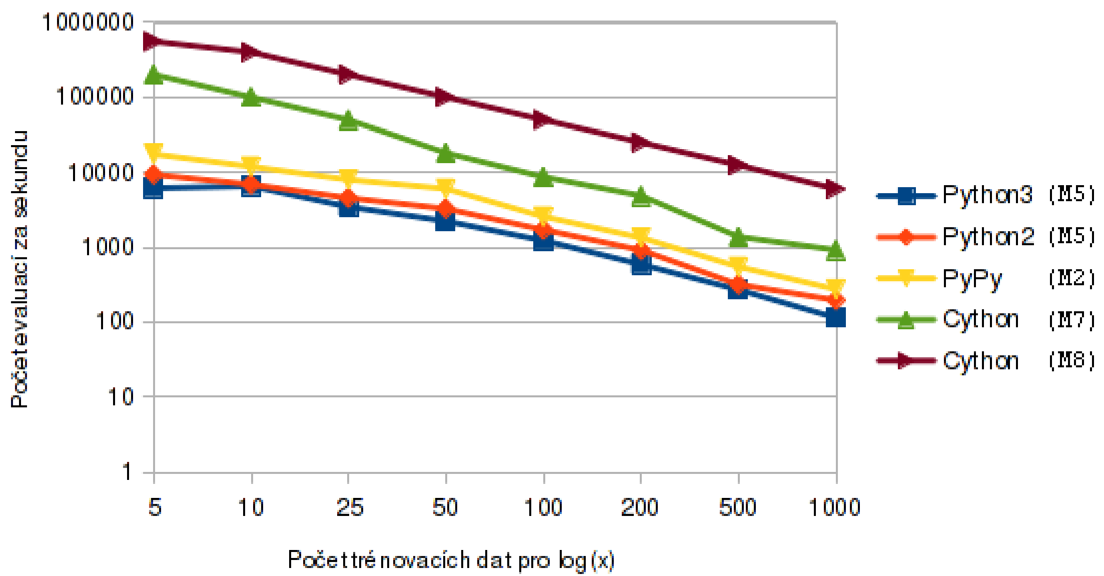
	Cython	C[11]
`5x5 MUL`	117644	119165
`6x6 MUL`	34780	40295

Tabulka 5.2: Počet evaluací za sekundu pro jednotlivé benchmarkové úlohy mezi Cythonem a jazykem C.



Obrázek 5.8: Výkonnost implementacích v Cythonu na 3 benchmarkových úlohách

Na grafu 5.9 můžete porovnat rychlost dvou implementací Cythonu s nejlepšími akceleračními technikami pro jednotlivé interprety.



Obrázek 5.9: Porovnání výkonnosti nejlepších akceleračních technik jednotlivých interpretů a dvou nejlepších metod v jazyce Cython.

5.5 Evoluční návrh klasifikátoru

Na závěr se zaměřím na vyhodnocení vlivu použití CGP v úloze klasifikace. Mimo jiné tento experiment ukazuje, jak lze využít CGP pro tvorbu vlastní aplikace.

Bylo vygenerováno 10 náhodných vektorů dat patřící do třídy A a 10 vektorů dat patřící do třídy B.

$$A = gauss(\mu = 5, \sigma = 3), gauss(\mu = 10, \sigma = 2)$$

$$B = gauss(\mu = 7, \sigma = 1), gauss(\mu = 13, \sigma = 2)$$

Výsledný chromozom funkce pak po transformaci vypadal:

```
float f(in_0, in_1)
{
    return (b2f((-b2f(in_0<in_1))>=b2f((3.14+3.14)<=in_0)) >= 0)? 1.0 : -1.0
}
```

kde funkce `b2f()` a `f2b()` přetypovávají datový typ `boolean` a datový typ `float`.

Úspěšnost při trénovaní dat byla 95% (10% miss rate, 0% false alarm). Ke klasifikaci bylo vygenerováno 2000 takových dat. Úspěšnost klasifikační funkce pak činila 76% (24% miss rate, 23% false alarm). Tato úspěšnost může být navýšena tím, použijeme-li více konstant.

Kapitola 6

Závěr

Cílem práce bylo akcelerovat kartézské genetické programování v Pythonu. Bylo využito několik způsobů implementace: tři způsoby JIT kompilací do bytecode, paralelní simulace a jazyk Cython. Všechny akcelerační metody byly porovnány na interpretech Python2, Python3 a PyPy.

Na základě experimentálních výsledků bylo zjištěno, že pomocí JIT kompilace můžeme algoritmus CGP výrazně zefektivnit. Záleží na způsobu, jakým byl kód zkompilován. Vestavěná funkce `compile()` je výhodná v případě, máme-li víc jak 100 trénovacích dat za použití Python3. V Pythonu2 se použití této vyplatí, bylo-li zadáno víc jak 10 trénovacích dat. Použití vlastní kompilace bytecode dopadla lépe, neboť funkce `compile()` provádí syntaktickou analýzu. Při použití kompilace do postfixové notace v případě symbolické regrese se původní algoritmus zrychlil 3.96 krát. Dále z výsledků vyplynulo, že paralelní výpočet zvyšuje počet evaluací za sekundu kandidátního řešení. Nevýhodou tohoto přístupu je jeho omezená aplikace. Ta je závislá na architektuře procesoru, na kterém běží výpočet.

Vůbec nejefektivnější akcelerací bylo vytvoření modulu pomocí jazyka Cython. Tento modul by se mohl v budoucí práci rozšířit o grafické uživatelské rozhraní. Dále by se do něj mohlo přidat jiné využití CGP např. návrh obrazových filtrů, evoluční umění atp. V budoucí práci bych se zaměřil na aplikaci CGP v jazyce C s využitím SIMD koprocesorů a Just-in-time kompilace. Dále by se mohl vytvořit rekonfigurovatelný obvod v FPGA (Field-programmable gate array), který by realizoval CGP. Jako cíl by bylo porovnání počtu evaluací za sekundu mezi CGP naimplementovaném v FPGA a v jazyce C využívající akcelerační techniky.

Literatura

[1] PyPy. [online], Naposledy navštíveno 5. 5. 2013.
URL http://pypy.org/

[2] Anderson, S. E.: Bit Twiddling Hacks. [online], Naposledy navštíveno 5. 5. 2013.
URL
http://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel

[3] Crow, O.: Efficient String Concatenation in Python. [online], Naposledy navštíveno 5. 5. 2013.
URL http://www.skymind.com/~ocrow/python_string/

[4] Dailey, J.: Python: range() vs. xrange(). [online], Naposledy navštíveno 5. 5. 2013.
URL http://justindailey.blogspot.cz/2011/09/python-range-vs-xrange.html

[5] Jelínek, J.; Zicháček, V.: *Biologie pro gymnázia*. Nakladatelství Olomouc, 2003, ISBN 80-7182-159-4.

[6] Martelli, A.; Ravenscroft, A.; Ascher, D.: *Python Cookbook*. O'Reilly Media, 2005, ISBN 978-0596007973.

[7] Miller, J.: *Cartesian Genetic Programming*. Springer, 2011, ISBN 978-3-642-17309-7.

[8] Oranchak, D.: Cartesian Genetic Programming for the Java Evolutionary Computing Toolkit (CGP for ECJ). [online], Naposledy navštíveno 5. 5. 2013.
URL http://oranchak.com/cgp/doc/#class

[9] Python Software Foundation: *Python v3.1.1 documentation*. 2013.
URL http://docs.python.org/3/

[10] Sekanina, L.; Vašíček, Z.; Bidlo, M.; aj.: *Evoluční hardware*. Academia, 2009, ISBN 80-200-1729-1.

[11] Vašíček, Z.: *Cartesian genetic programming*. 2013.
URL http://www.fit.vutbr.cz/~vasicek/cgp/

[12] Vašíček, Z.; Slaný, K.: Efficient Phenotype Evaluation in Cartesian Genetic Programming. In *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012, LNCS*, ročník 7244, editace A. Moraglio; S. Silva; K. Krawiec; P. Machado; C. Cotta, Malaga, Spain: Springer Verlag, 11-13 Duben 2012, s. 266–278, doi:doi:10.1007/978-3-642-29139-5_23.

Příloha A

Obsah CD

- /**tex** – textová část dokumentace
- /**cgp** – zdrojové kódy výsledného modulu
- /**src** – zdrojové kódy s testy
  - /**cython** – implementace v jazyce cython
    - /**v0.1** – využívající dynamické typování
    - /**v0.2** – využívající statické typování
    - /**v0.3** – využívající statické typování s rozbalenými funkcemi
  - /**data** – sada benchmarkových úloh
  - /**v0.1** – bez výpočtu fitness funkce
  - /**v0.2** – využívající paralelní simulaci
  - /**v0.3** – využívající rozdělení v/v dat do trénovacích vektorů
  - /**v0.4** – JIT kompilace pomocí funkce `compile()`
  - /**v0.5** – JIT kompilace do bytecode
  - /**v0.6** – JIT kompilace do bytecode v postfixové notaci
- /**viewer** – prohlížeč chromozómů

Příloha B

Bytecode

Tato část přílohy se zabývá funkcí `CodeType` modulu `types`. Pomocí ní může být vytvořen bytecode. Ovšem funkce `CodeType()` je v dokumentaci Pythonu [9] popsána velice stručně. Číselné hodnoty bytecode pak nejsou zdokumentovány vůbec, seznam použitých bytecode instrukcí nalezenete na další stránce.

```
>>> print(types.CodeType.__doc__)
code(argcount, kwonlyargcount, nlocals, stacksize, flags, codestring,
      constants, names, varnames, filename, name, firstlineno,
      lnotab[, freevars[, cellvars]])

Create a code object.  Not for the faint of heart.
```

- **argcount** – počet argumentů kódu
- **kwonlyargcount** – počet klíčových slov argumentu[1]
- **nlocals** – počet lokálních proměnných
- **stacksize** – velikost zásobníku interpretu
- **flags** – bitové pole obsahující informace o daném kódu
- **codestring** – bytecode implementováno jako pole bytů
- **constants** – konstanty kódu; uloženo ve formátu `tuple`
- **names** – jména lokálních proměnných; uloženo ve formátu `tuple`
- **varnames** – jména argumentů kódu; uloženo ve formátu `tuple`
- **filename** – název souboru, ve kterém bude kód uložen
- **name** – jméno, kterým se definuje tento kód
- **firstlineno** – číslo prvního řádku v zdrojovém kódu Pythonu
- **lnotab** – mapování čísel řádku kódu do bytecode; uloženo ve formátu `tuple`

[1] v Pythonu2 tento argument není

<table>
<tr><th>Bytecode</th><th>Název instrukce</th><th>Sémantika</th><th>Operace nad zásobníkem</th></tr>
<tr><td>1</td><td>POP_TOP</td><td>Vyjmutí proměnné z vrcholu zásobníku.</td><td></td></tr>
<tr><td>9</td><td>NOP</td><td>Nedělá žádnou operaci.</td><td></td></tr>
<tr><td>15</td><td>UNARY_INVERT</td><td>Binárně neguje vrchol zásobníku.</td><td>TOS = ~TOS.</td></tr>
<tr><td>20</td><td>BINARY_MULTIPLY</td><td>Násobení.</td><td>TOS = TOS1 * TOS</td></tr>
<tr><td>23</td><td>BINARY_ADD</td><td>Sčítání.</td><td>TOS = TOS1 + TOS</td></tr>
<tr><td>24</td><td>BINARY_SUBSTRACT</td><td>Odčítání.</td><td>TOS = TOS1 - TOS</td></tr>
<tr><td>27</td><td>BINARY_TRUE_DIVIDE</td><td>Dělení.</td><td>TOS = TOS1 / TOS</td></tr>
<tr><td>64</td><td>BINARY_AND</td><td>Logický součin.</td><td>TOS = TOS1 & TOS</td></tr>
<tr><td>65</td><td>BINARY_XOR</td><td>Logický exklusivní součet.</td><td>TOS = TOS1 ^ TOS</td></tr>
<tr><td>66</td><td>BINARY_OR</td><td>Logický součet.</td><td>TOS = TOS1 | TOS</td></tr>
<tr><td>25</td><td>BINARY_SUBSCR</td><td>Přístup k hodnotě pole.</td><td>TOS = TOS1[TOS]</td></tr>
<tr><td>60</td><td>STORE_SUBSCR</td><td>Uložení hodnoty do pole.</td><td>TOS1[TOS] = TOS2</td></tr>
<tr><td>90</td><td>LOAD_NAME</td><td>Uložení hodnoty na vrcholu zásobníku do proměnné.</td><td>name = TOS</td></tr>
<tr><td>100</td><td>LOAD_CONST</td><td>Nahrání konstanty.</td><td>push(co_constants[arg])</td></tr>
<tr><td>101</td><td>LOAD_NAME</td><td>Nahrání proměnné.</td><td>push(co_names[arg])</td></tr>
<tr><td>83</td><td>RETURN_VALUE</td><td>Vrátí hodnotu, která je na vrcholu zásobníku.</td><td>return TOS</td></tr>
</table>

Tabulka B.1: Použité instrukce bytecode

Příloha C

Tabulky výsledků

V této části přílohy naleznete tabulky dosažených výsledků pro benchmarkové úlohy.
Parametry CGP pro návrh kombinačních obvodů: řádků=1, sloupců=40, levelback=40, funkce=(id, not, and, or, xor, nand, nor, xnor)
Parametry CGP pro symbolickou regresi: řádků=1, sloupců=15, levelback=15, funkce=(id, 1.0, 0.5, 0.25, add, sub, mul, div, sin)
CPU params: Intel Core i3-2310M (3MB Cache, 2.10 GHz) OS: Arch Linux 64bit

Problém	TV[1]	M0	M1	M2	M3	M4	M5
3x3 ADD	1	12398	8264	8857	1432	6529	5057
4x4 ADD	4	11290	7865	4290	1361	4616	3877
5x5 ADD	16	8381	6053	1229	787	1814	1704
6x6 ADD	64	4026	3788	327	337	544	551
4x4 MUL	4	10439	7137	3479	1017	3638	2953
5x5 MUL	16	7541	5099	1075	732	1458	1366
6x6 MUL	64	3540	2400	236	238	359	368
PARITY	8	12508	8665	4876	3876	6833	7095
LOG(x)	5	14246	-	5228	3562	5390	6162
LOG(x)	10	12545	-	4314	1990	5637	6559
LOG(x)	25	9279	-	1928	1427	3155	3465
LOG(x)	50	6363	-	991	921	1836	2231
LOG(x)	100	3984	-	543	630	1069	1226
LOG(x)	200	2268	-	195	337	422	587
LOG(x)	500	998	-	110	166	230	270
LOG(x)	1000	506	-	34	87	80	114
COS(x)	1000	519	-	63	82	130	143
SIN(x)	1000	519	-	58	83	125	141
LOG(x)	1000	506	-	34	87	80	114
XXYZ(x)	1000	524	-	68	127	140	144

Tabulka C.1: Počet evaluací za sekundu pro jednotlivé benchmarkové úlohy v Pythonu3

Problém	TV	M0	M1	M2	M3	M4	M5
3x3 ADD	1	13694	7297	9426	1700	7342	5150
4x4 ADD	4	12611	6952	4306	1559	4961	3965
5x5 ADD	16	10108	5021	1221	907	1939	1783
6x6 ADD	64	5246	2173	355	442	583	595
4x4 MUL	4	11580	6193	3709	1349	4127	3152
5x5 MUL	16	8929	3877	954	719	1424	1252
6x6 MUL	64	4514	1472	239	276	374	376
PARITY	8	13829	6678	5471	4911	7779	7987
LOG(x)	5	18149	-	7250	2991	8406	9347
LOG(x)	10	16480	-	4354	4525	6469	6915
LOG(x)	25	12595	-	1993	3327	3889	4555
LOG(x)	50	8928	-	1249	2156	2695	3269
LOG(x)	100	5686	-	564	1300	1358	1708
LOG(x)	200	3319	-	286	521	737	915
LOG(x)	500	1472	-	114	315	304	317
LOG(x)	1000	758	-	58	114	154	196
COS(x)	1000	762	-	40	132	115	176
SIN(x)	1000	762	-	43	132	122	188
LOG(x)	1000	758	-	58	114	154	196
XXYZ(x)	1000	758	-	52	130	141	193

Tabulka C.2: Počet evaluací za sekundu pro jednotlivé benchmarkové úlohy v Pythonu2

Problém	TV	M0	M1	M2	M3	M4	M5
3x3 ADD	1	20456	24105	12130	858	8429	6502
4x4 ADD	4	32458	15016	8564	763	4357	4173
5x5 ADD	16	42089	9958	2825	402	1306	1380
6x6 ADD	64	17794	3562	803	184	349	368
4x4 MUL	4	51083	16538	9315	677	4447	4297
5x5 MUL	16	37742	7810	2152	341	1121	1168
6x6 MUL	64	17275	2262	511	137	277	295
PARITY	8	61059	17996	22319	2301	4450	4338
LOG(x)	5	39520	-	17521	1370	6277	6253
LOG(x)	10	41085	-	11941	1660	7955	3902
LOG(x)	25	40903	-	7982	1037	4570	1869
LOG(x)	50	25739	-	6048	621	2873	998
LOG(x)	100	14389	-	2563	364	1311	490
LOG(x)	200	7714	-	1339	156	666	246
LOG(x)	500	3091	-	547	81	268	46
LOG(x)	1000	1674	-	274	33	136	12
COS(x)	1000	1625	-	158	37	100	20
SIN(x)	1000	1617	-	175	36	107	5
LOG(x)	1000	1674	-	274	33	136	11
XXYZ(x)	1000	1623	-	224	36	123	8

Tabulka C.3: Počet evaluací za sekundu pro jednotlivé benchmarkové úlohy v PyPy

Problém	TV	M6	M7	M8
3x3 ADD	1	9033	200000	1250000
4x4 ADD	4	4957	66666	606060
5x5 ADD	16	1732	18180	188676
6x6 ADD	64	481	4876	45556
4x4 MUL	4	4327	50000	333332
5x5 MUL	16	1523	15384	117644
6x6 MUL	64	434	3772	34780
PARITY	8	4493	200000	666642
LOG(x)	5	5569	200000	555552
LOG(x)	10	3334	100000	400000
LOG(x)	25	1554	50000	202020
LOG(x)	50	809	18180	101052
LOG(x)	100	413	8692	50540
LOG(x)	200	210	4876	25242
LOG(x)	500	84	1376	12500
LOG(x)	1000	43	920	6000
COS(x)	1000	42	1500	12000
SIN(x)	1000	43	2400	12000
LOG(x)	1000	43	920	6000
XXYZ(x)	1000	80	1712	12000

Tabulka C.4: Počet evaluací za sekundu pro jednotlivé benchmarkové úlohy v Cythonu

Legenda

M0 – Bez žádného výpočtu fitness funkce.
M1 – Využití techniky paralelní simulace.
M2 – Rozdělení vstupních a výstupních dat do trénovacích vektorů.
M3 – JIT kompilace pomocí funkce `compile()`.
M4 – JIT kompilace do bytecode.
M5 – JIT kompilace do bytecode v postfixové notaci.
M6 – Cython s dynamickým typováním.
M7 – Cython se statickým typováním.
M8 – Cython se statickým typováním a rozbalenými smyčkami.
TV – Počet trénovacích metod.

Příloha D

Použití výsledného modulu

Po rozbalení daného archívu zkompilujte modul pomocí příkazu `make py2` pro interpret Pythonu2. Pro interpret Pythonu3 spusťte příkaz `make py3` či jednoduše `make`.

V interpretu Pythonu daný modul importujte. Tento modul obsahuje třídu `Cgp`. Pro zobrazení nápovědy můžete v interpretu Pythonu napsat příkaz `print(Cgp.__doc__)`.

Syntaxe a sémantika tohoto modulu by měla být podobná i pro moduly napsané v Pythonu. Jediné dvě věci, ve kterých se liší je změna funkční množiny (viz 4.1) a přístup k výslednému chromozómu (přes atribut `chromosome`).

D.1 Konstruktor

```
__init__(rows, cols, lback, in, out)
```

Konstruktor vytvoří objekt dané třídy. Jeho parametry jsou nepovinné: počet řádků grafu (1), počet sloupců grafu (40), míra propojení uzlu (40), vstupní data, výstupní data. V závorkách jsou uvedeny jejich implicitní hodnoty. Vstupní a výstupní data v případě návrhu kombinačních obvodů jsou repreztovány jako seznam celých čísel `[in1, in2, in3]`, kde `in1`, `in2`, `in3` jsou jednotlivé vstupy (či výstupy) genetického programování. V případě symbolické regrese jsou data reprezentována seznamem dimenzí, kde jednotlivá dimenze představuje seznam reálných čísel `[in1, in2, in3]`, kde `in1`, `in2`, `in3` jsou seznamy reálných čísel. Musí platit podmínka, aby počet vstupních dimenzí byl všude stejný. Tedy musí platit rovnice

$$len(in_1) = len(in_2) = len(in_3) = \ldots = len(in_i),$$

kde i je celkový počet dimenzí. Podobná podmínka musí platit i pro výstup.

D.2 Změna grafu

```
graph(rows, cols, lback)
```

Tato metoda manipuluje s grafem CGP. Příjimá dva povinné parametry: počet řádků a počet sloupců. Jako třetí parametr může být zadána míra propojení mezi jednotlivými sloupci (level back). Nebyl-li zadán, vybere se maximální propojení grafu.

D.3 Čtení dat ze souboru

`file(filename)`

Pomocí této metody můžeme zadat jméno souboru, ve kterém jsou uložena data pro genetické programování. Formát souboru je následující:

- komentář je oddělen znakem #
- ignorují se prázdné řádky
- v případě symbolické regrese jsou jednotlivé dimenze odděleny mezerou
- v případě návrhu kombinačních obvodů se mezery ignorují
- formát řádku: vstup je od výstupu oddělen znakem dvojtečky `vstup : výstup`
- počet vstupů a výstupů se odvodí z prvního řádku s daty

D.4 Změna dat

`data(input, output)`

Tato metoda načte data v podobném formátu jako v konstruktoru. Příjimá dva parametry – vstupní data, výstupní data.

D.5 Spuštění algoritmu CGP pro symblockou regresi či klasifikaci

```
run(generation, population, mutations, acc, runs)
run_classification(generation, population, mutations, acc, runs)
```

Vstupní parametry jsou: počet generací, velikost populace, počet mutací, dále je nutná přesnost. Nepovinný parametr je počet běhů genetického programování, jehož vychozí hodnota je jedna.

D.6 Spuštění algoritmu CGP pro návrh kombinačních obvodů

`run(generation, population, mutations, runs)`

Vstupní parametry jsou: počet generací, velikost populace, počet mutací. Nepovinný parametr je počet běhů genetického programování, jehož vychozí hodnota je jedna.

D.7 Změna funkční množiny

Objekt třídy `Cgp` obsahuje atribut `function` tento atribut představuje danou funkční množinu. Jednotlivé funkční množiny můžeme měnit pomocí vestavěných funkcí. Pro obvody jsou k dispozici tyto vestavěné funkční množiny:

- `allLogicalOperations()` – (id, and, or, xor, not, nand, nor, nxor)
- `booleanAlgebra()` – (id, and, or, not)
- `reedMuller()` – (id, or, xor, not)
- `moje()` – (id, or, xor, and)
- `nandOnly()` – (id, nand)

Pro symbolickou regresi:

- `symbolicRegression()` – (0.25, 0.50, 1.00, id, add, sub, mul, div)
- `symbolicRegressionWithSin()` – (0.25, 0.50, 1.00, id, add, sub, mul, div, sin)

Pro klasifikaci:

- `classification()` – (id, add, sub, mul, div, lt, lte, gt, gte, eq, and, or, not, nor, nand, neg, if, iflez, 1.24, 2.71, 3.14)

Například pomocí příkazu `cgp.function.booleanAlgebra()` změníme funkční množinu na Booleovu algebru. Nebyla-li zadána funkční množina, vybere se podle typu dat buď `allLogicalOperations()`, nebo `symbolicRegressionWithSin()`. V případě spuštění klasifikace pomocí metody `run_classification()` se vybere funkční množina určená pro klasifikaci. Chceme-li vytvořit svou funkční množinu, musíme objektu `function` přidat tyto atributy:

- `type` – typ funkční množiny
- `set` – seznam funkcí
- `arity` – arita jednotlivých funkcí
- `table` – názvy funkcí, které se použijí při tisku chromozómu

Implementace nepodporuje aritu funkcí větší než dva. V případě klasifikace větší než tři.

D.8 Výsledky algoritmu CGP

- `resultChrom()` – výsledkem této metody je chromozóm ve formě řetězce, jenž může být využit v CGPTools
- `showChrom()` – výsledkem této metody je chromozóm, který je v čitelnější formě
- `bestFitness` – hodnota odpovídající nejlepší nalezené fitness hodnoty
- `chrom` – seznam celých čísel představující chromozóm
- `elapsed` – doba běhu CGP algoritmu
- `evalspersec` – počet evaluací fitness za sekundu